I N F S Y S
R E S E A R C H
R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME
ARBEITSBEREICH WISSENSBASIERTE SYSTEME

19TH INTERNATIONAL CONFERENCE ON APPLICATIONS OF DECLARATIVE PROGRAMMING AND KNOWLEDGE MANAGEMENT (INAP 2011)
AND
25TH WORKSHOP ON LOGIC PROGRAMMING (WLP 2011)

VIENNA, SEPTEMBER 28-30, 2011
PROCEEDINGS

Salvador Abreu Johannes Oetsch Jörg Pührer Dietmar Seipel
Hans Tompits Masanobu Umeda Armin Wolf (eds.)

INFSYS RESEARCH REPORT 1843-11-06
SEPTEMBER 2011

Institut für Informationssysteme
Arbeitsbereich
Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-11-06, SEPTEMBER 2011

PROCEEDINGS OF THE 19TH INTERNATIONAL CONFERENCE ON APPLICATIONS OF DECLARATIVE PROGRAMMING AND KNOWLEDGE MANAGEMENT (INAP 2011)

AND

25TH WORKSHOP ON LOGIC PROGRAMMING (WLP 2011)

VIENNA, AUSTRIA, SEPTEMBER 28-30, 2011

Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel,
Hans Tompits, Masanobu Umeda, and Armin Wolf[1]
(Volume Editors)

[1] Editors' address: Salvador Abreu, Departamento de Informática Universidade de Évora, R. Romão Ramalho, 59, P-7000 Évora, Portugal, e-mail: spa@di.fct.unl.pt.
Johannes Oetsch, Institut für Informationssysteme, Arbeitsbereich Wissensbasierte Systeme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria, e-mail: oetsch@kr.tuwien.ac.at.
Jörg Pührer, Institut für Informationssysteme, Arbeitsbereich Wissensbasierte Systeme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria, e-mail: puehrer@kr.tuwien.ac.at.
Dietmar Seipel, Institut für Informatik, Julius-Maximilians-Universität Würzburg, Am Hubland, D-97074 Würzburg, Germany, e-mail: seipel@informatik.uni-wuerzburg.de.
Hans Tompits, Institut für Informationssysteme, Arbeitsbereich Wissensbasierte Systeme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria, e-mail: tompits@kr.tuwien.ac.at.
Masanobu Umeda, Department of Creative Informatics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka, 820-8502, Japan, e-mail: umerin@ci.kyutech.ac.jp.
Armin Wolf, Fraunhofer FIRST, Kekuléstraße 7, D-12489 Berlin, Germany, e-mail: armin.wolf@first.fraunhofer.de.

Preface

This volume consists of the contributions presented at the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011), which were held at Hotel Castle Wilheminenberg, Vienna, Austria, from September 28 to 30, 2011.

INAP is a communicative and dense conference for intensive discussion of applications of important technologies around logic programming, constraint problem solving, and closely related computing paradigms. It comprehensively covers the impact of programmable logic solvers in the internet society, its underlying technologies, and leading edge applications in industry, commerce, government, and societal services.

The series of workshops on (constraint) logic programming brings together researchers interested in logic programming, constraint programming, and related areas like databases and artificial intelligence. Previous workshops have been held in Germany, Austria, Switzerland, and Egypt, serving as the annual meeting of the Society of Logic Programming (GLP, Gesellschaft für Logische Programmierung e.V.).

Following the success of previous occasions, INAP and WLP were this year again jointly organised in order to promote the cross-fertilisation of ideas and experiences among researches and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming languages and related areas.

Both events received a total of 35 submissions from authors of 16 countries (Austria, Belgium, Canada, Czech Republic, Egypt, Finland, France, Germany, India, Italy, Japan, Lebanon, Portugal, Slovakia, Tunisia, and the United States). Each submission was assigned to three members of the PC for reviewing and 27 submissions were accepted for presentation. Besides technical contributions, the program includes also system descriptions and application papers. More specifically, for INAP, the program comprises nine technical contributions, two application papers, and four system descriptions, whilst the contributions for WLP constitute six research papers and five system descriptions. Additionally, the program includes also two invited talks, given by Stefan Szeider and Michael Fink (both from the Vienna University of Technology, Austria).

In concluding, I would like to thank all authors for their submissions and all members of the program committee, as well as all additional referees, for the time and effort spent on the careful reviewing of the papers. Furthermore, special thanks go to the members of the organising committee, Johannes Oetsch, Jörg Pührer, and Eva Nedoma, who were indispensable towards the realisation of the event. Last but not least, I am grateful to the Kurt-Gödel-Society for financially supporting the event. Excelsior!

Vienna, September 2011

Hans Tompits, Conference Chair

Organisation

INAP 2011

Conference Chair

Hans Tompits (*Vienna University of Technology*)

Track Chairs

Salvador Abreu (*Universidade de Évora*)
Dietmar Seipel (*University of Würzburg*)
Hans Tompits (*Vienna University of Technology*)
Masanobu Umeda (*Kyushu Institute of Technology*)
Armin Wolf (*Fraunhofer FIRST*)

Program Committee

Salvador Abreu (*Universidade de Évora*)
José Alferes (*Universidade Nova de Lisboa*)
Sergio Alvarez (*Boston College*)
Grigoris Antoniou (*University of Crete*)
Marcello Balduccini (*Kodak Research Labs*)
Chitta Baral (*Arizona State University*)
Christoph Beierle (*FernUniversität in Hagen*)
Philippe Besnard (*Université Paul Sabatier*)
Stefan Brass (*University of Halle*)
Gerd Brewka (*University of Leipzig*)
Philippe Codognet (*University of Tokyo*)
Vitor Santos Costa (*Universidade do Porto*)
James P. Delgrande (*Simon Fraser University*)
Marc Denecker (*Katholieke Universiteit Leuven*)
Marina De Vos (*University of Bath*)
Daniel Diaz (*University of Paris 1*)
Jürgen Dix (*Clausthal University of Technology*)
Esra Erdem (*Sabanci University*)
Gerhard Friedrich (*Alpen-Adria-Universität Klagenfurt*)
Michael Fink (*Vienna University of Technology*)
Thom Frühwirth (*University of Ulm*)
Johannes Fürnkranz (*Technische Universität Darmstadt*)
Michael Gelfond (*Texas Tech University*)
Carmen Gervet (*German University in Cairo*)
Ulrich Geske (*University of Potsdam*)
Gopal Gupta (*University of Texas at Dallas*)
Petra Hofstedt (*Brandenburg University of Technology Cottbus*)
Anthony Hunter (*University College London*)
Katsumi Inoue (*National Institute of Informatics*)
Tomi Janhunen (*Aalto University*)

Gabriele Kern-Isberner (*University of Dortmund*)
Nicola Leone (*University of Calabria*)
Vladimir Lifschitz (*University of Texas at Austin*)
Alessandra Mileo (*National University of Ireland*)
Ulrich Neumerkel (*Vienna University of Technology*)
Ilkka Niemelä (*Aalto University*)
Vitor Nogueira (*Universidade de Évora*)
David Pearce (*Universidad Politécnica de Madrid*)
Reinhard Pichler (*Vienna University of Technology*)
Axel Polleres (*National University of Ireland*)
Enrico Pontelli (*New Mexico State University*)
Irene Rodrigues (*Universidade de Évora*)
Carolina Ruiz (*Worcester Polytechnic Institute*)
Torsten Schaub (*University of Potsdam*)
Dietmar Seipel (*University of Würzburg*)
V.S. Subrahmanian (*University of Maryland*)
Terrance Swift (*Universidade Nova de Lisboa*)
Hans Tompits (*Vienna University of Technology*)
Masanobu Umeda (*Kyushu Institute of Technology*)
Kewen Wang (*Griffith University*)
Emil Weydert (*University of Luxembourg*)
Armin Wolf (*Fraunhofer FIRST*)
Osamu Yoshie (*Waseda University*)

WLP 2011

Program Chair

Hans Tompits (*Vienna University of Technology*)

Program Committee

Slim Abdennadher (*German University Cairo*)
Gerd Brewka (*University of Leipzig*)
Christoph Beierle (*FernUniversität in Hagen*)
François Bry (*University of Munich*)
Marc Denecker (*Katholieke Universiteit Leuven*)
Marina De Vos (*University of Bath*)
Jürgen Dix (*Clausthal University of Technology*)
Esra Erdem (*Sabanci University*)
Wolfgang Faber (*University of Calabria*)
Michael Fink (*Vienna University of Technology*)
Thom Frühwirth (*University of Ulm*)
Carmen Gervet (*German University in Cairo*)
Ulrich Geske (*University of Potsdam*)
Michael Hanus (*Christian-Albrechts-University of Kiel*)
Petra Hofstedt (*Brandenburg University of Technology Cottbus*)
Steffen Hölldobler (*Dresden University of Technology*)
Tomi Janhunen (*Aalto University*)
Ulrich John (*SIR Dr. John UG*)

Local Organisation

Additional Referees

Table of Contents

WLP System Descriptions I

INAP Technical Papers II: Answer-Set Programming and Abductive Reasoning

WLP System Descriptions II

INAP System Descriptions II

INAP Application Papers I

WLP Technical Papers II: Answer-Set Programming and Model Expansion

INAP Application Papers II

INAP Technical Papers III: Semantics

The Parameterized Complexity of Constraint Satisfaction and Reasoning*

Stefan Szeider[1]

Institute of Information Systems, Vienna University of Technology, A-1040 Vienna, Austria
stefan@szeider.net

Abstract. Parameterized Complexity is a new and increasingly popular theoretical framework for the analysis and algorithmic solution of NP-hard problems. The framework allows to take structural properties of problem instances into account and supports a more fine-grained analysis than the traditional complexity framework. We outline some of the basic concepts of Parameterized Complexity and indicate some recent results on problems arising in Constraint Satisfaction and Reasoning.

1 Introduction

Computer science has been quite successful in devising fast algorithms for important computational tasks, for instance, to sort a list of items or to match workers to machines. By means of a theoretical analysis one can guarantee that the algorithm will always find a solution quickly. Such a worst-case performance guarantee is the ultimate aim of algorithm design. The traditional theory of algorithms and complexity as developed in the 1960s and 1970s aims at performance guarantees in terms of one dimension only, the input size of the problem. However, for many important computational problems that arise from real-world applications the traditional theory cannot give reasonable (i.e., polynomial) performance guarantees. The traditional theory considers such problems as *intractable*. Nevertheless, heuristics-based algorithms and solvers work surprisingly well on real-world instances of such problems. Take for example the satisfiability problem (SAT) of propositional reasoning. No algorithm is known that can solve a SAT instance on n variables in $2^{o(n)}$ steps (the widely believed Exponential Time Hypothesis states that such an algorithm is impossible [23]). On the other hand, state-of-the-art SAT solvers solve routinely instances with hundreds of thousands of variables in a reasonable amount of time (see e.g., [17]). Hence there is an enormous gap between theoretical performance guarantees and the empirically observed performance of solvers. This gap separates theory-oriented and applications-oriented research communities.

Parameterized Complexity is a new theoretical framework for the analysis and algorithmic solution of NP-hard problems. It offers a great potential for reducing the theory-practice gap. The key idea is to consider—in addition to the input size—a secondary dimension, the parameter, and to design and analyse algorithms in this two-dimensional setting. Virtually in every conceivable context we know more about the input data than just its size in bytes. The second dimension (the parameter) can represent this additional information. This two-dimensional setting gives raise to a foundational theory of algorithms and complexity that can be closer to the problems as they appear in the real world.

Parameterized Complexity has been introduced and pioneered by R. Downey and M. R. Fellows [6] and is receiving growing interest as reflected by the recent publication of two further monographs [10, 30] and hundreds of research papers (see the references in [6, 10, 30]). In more and more research areas such as Computational Biology or Computational Geometry the merits of Parameterized Complexity become apparent (see, e.g., [16, 20]).

* Invited talk at INAP 2011/WLP 2011 (The 19th International Conference on Applications of Declarative Programming and Knowledge Management, and The 25th Workshop on Logic Programming). Research supported by the European Research Council, grant reference 239962 (COMPLEX REASON).

2 Parameterized Complexity: Basic Concepts and Definitions

In the following we outline the central concepts of Parameterized Complexity.

An instance of a parameterized problem is a pair (I, k) where I is the *main part* and k is the *parameter*; the latter is usually a non-negative integer. The central notion of the field is *fixed-parameter tractability* (FPT) which refers to solvability in time $f(k)n^c$, where f is some (possibly exponential) function of the parameter k, c is a constant, and n denotes the size of the instance with respect to some reasonable encoding. A fixed-parameter tractable problem can therefore be solved in polynomial time for any fixed value of the parameter, and, importantly, the order of the polynomial does not depend on the parameter. This is significantly different from problems that can be solved in, say, time n^k, which also gives polynomial-time solvability for each fixed value of k, but since the order of the polynomial depends on k it does not scale well in k and quickly becomes inefficient for small values of k.

Take for example the VERTEX COVER problem: Given a graph and an integer k, the question is whether there is a set of k vertices such that each edge of the graph has at least one of its ends in this set. The problem is NP-complete, but fixed-parameter tractable for parameter k. Currently the best known fixed-parameter algorithm for this problem runs in time of order $1.2738^k + kn$ [4]. This algorithm is practical for huge instances as long as the parameter k is below 100. The situation is dramatically different for the INDEPENDENT SET problem, where for a given graph and an integer k it is asked whether there is a set of k vertices such that no edge joints two vertices in the set. Also this problem is NP-complete, and indeed for traditional complexity the problems VERTEX COVER and INDEPENDENT SET are essentially the same, as there is a trivial polynomial-time transformation from one problem to the other (the complement set of a vertex cover is an independent set and vice versa). However, no fixed-parameter algorithm for INDEPENDENT SET is known and the Parameterized Complexity of this problem appears to be very different from the complexity of VERTEX COVER. Theoretical evidence suggests that INDEPENDENT SET cannot be solved significantly faster than by trying all subsets of size k, which gives a running time of order n^k.

The subject of Parameterized Complexity splits into two complementary questions, each with its own mathematical toolkit and methods:

1. How to design and improve fixed-parameter algorithms for parameterized problems. For this question there exists a rich *toolkit of algorithmic techniques* (see, e.g., [41]).

2. How to gather evidence that a parameterized problem is not fixed-parameter tractable. For this question a *completeness theory* has been developed which is similar to the theory of NP-completeness (see, e.g., [5]) and allows the accumulation of strong theoretical evidence that a parameterized problem is *not* fixed-parameter tractable.

3 How to Parameterize?

Most research in Parameterized Complexity considers optimization problems, where the parameter is a bound on the objective function, also called solution size. For instance, the standard parameter for VERTEX COVER is the size of the vertex cover we are looking for. However, many problems that arise in Constraint Satisfaction and Reasoning are not optimization problems, and it seems more natural to consider parameters that indicate the presence of a "hidden structure" in the problem instance. It is a widely accepted view that the hidden structure of real-world problem instances is of high significance for empirical problem-hardness.

3.1 Backdoors

If a computational problem is intractable in general, it is a natural question to ask for subproblems for which the problem is solvable in polynomial-time, and indeed much research has been devoted to this question. Such tractable subproblems are sometimes called "islands of tractability" or "tractable fragments." It seems unlikely that a problem instance originating from a real-world application belongs to one of the known tractable fragments, but it might be "close" to one. The concept of *backdoor sets* offers a generic way to gradually enlarge and extend a tractable subproblem and thus to solve problem instances efficiently if they are close to a tractable fragment. The size of a smallest backdoor set indicates the distance between an instance and a tractable fragment. Backdoor sets were introduced in the context of propositional and

constraint-based reasoning [45] but similar notions can be defined for other reasoning problems. Roughly speaking, after eliminating the variables of a backdoor set one is left with an instance that belongs to the tractable subproblem under consideration. The "backdoor approach" to reasoning problems involves two tasks. The first task is to detect a small backdoor set by a fixed-parameter algorithm, parameterized by the size of the backdoor set. The second task is to solve the reasoning problem efficiently using the information provided by the backdoor set.

There are several Parameterized Complexity results on backdoor sets for the SAT problem, including [31, 42, 36], but also for problems beyond NP such as Model Counting and QBF-Satisfiability [32, 37]. Very recently a backdoor approach has been developed for Answer Set Programming and Abstract Argumentation [9, 34].

3.2 Decompositions

A key technique for coping with hard computational problems is to decompose the problem instance into small tractable parts, and to reassemble the solutions of the parts to a solution of the entire instance. One aims at decompositions for which the overall complexity depends on how much the parts overlap, the "width" of the decomposition. The most popular and widest studied decomposition method is *tree decomposition* with the associated parameter *treewidth*. A recent survey by Hlinený et al. covers several decomposition methods with particular focus on fixed-parameter tractability [22].

Recent results on the Parameterized Complexity of reasoning problems with respect to decomposition width include results on Disjunctive Logic Programming and Answer-Set Programming with weight constraints [18, 35], Abductive Reasoning [19], Satisfiability and Propositional Model Counting [39, 33], Constraint Satisfaction and Global Constraints [40, 38], and Abstract and Value-Based Argumentation [7, 24].

3.3 Locality

Practical algorithms for hard reasoning problems are often based on *local search* techniques. The basic idea is to start with an arbitrary candidate solution and to try to improve it step by step, at each step moving from one candidate solution to a better "neighbor" candidate solution. It would provide an enormous speed-up if one could perform k elementary steps of local search efficiently in one "giant" k-step. Such a giant k-step also decreases the probability of getting stuck at a poor local optimum. However, the obvious strategy for performing one giant k-step requires time of order N^k (assuming a candidate solution has N neighbour solutions), which is impractical already for very small values of k since typically N is related to the input size. A challenging objective is the design of fixed-parameter algorithms (with respect to parameter k) that compute a giant k-step. Recent work on parameterized local search includes the problem of minimizing the Hamming weight of satisfying assignments for Boolean CSP [25], and for the MAX SAT problem [44].

Local consistency is a further form of locality that plays an important role in constraint satisfaction and is one of the oldest and most fundamental concepts of in this area. It can be traced back to Montanari's famous 1974 paper [29]. If a constraint network is locally consistent, then consistent instantiations to a small number of variables can be consistently extended to any further variable. Hence local consistency avoids certain dead-ends in the search tree, in some cases it even guarantees backtrack-free search [1, 13]. The simplest and most widely used form of local consistency is arc-consistency, introduced by Mackworth [26], and later generalized to k-consistency by Freuder [12]. A constraint network is *k-consistent* if each consistent assignment to $k-1$ variables can be consistently extended to any further k-th variable. It is a natural question to ask for the Parameterized Complexity of checking whether a constraint network is k-consistent, taking k as the parameter. This question has been subject to a recent study [15].

3.4 Above or Below Guaranteed Bounds

For some optimization problems that arise in constraint satisfaction and reasoning, the standard parameter (solution size) is not a very useful one. Take for instance the problem MAX SAT. The standard parameter is the number of satisfied clauses. However, it is well-known that one can always satisfy at least half of the

clauses. Hence, if we are given m clauses, and if we want to satisfy at least k of them, then the answer is clearly yes if $k \leq m/2$. On the other hand, if $k > m/2$ then $m < 2k$, hence the size of the given formula is bounded in terms of the parameter k, and thus can be trivially solved by brute force in time that only depends on k. Less trivial is the question of whether we can satisfy at least $m/2 + k$ clauses, where k is the parameter. Such a problem is called *parameterized above a guaranteed value* [27, 28]. Over the last few years, several variants of MAX SAT but also optimization problems regarding ordering constraints have been studied, parameterized above a guaranteed value. A recent survey by Gutin and Yeo covers these results [21].

4 Kernelization: Preprocessing with Guarantee

Preprocessing and data reduction are powerful ingredients of virtually every practical solver. Before performing a computationally expensive case distinction, it seems always better to seek for a "safe step" that simplifies the instance, and to preprocess. Indeed, the success of practical solvers relies often on powerful preprocessing techniques. However, preprocessing has been neglected by traditional complexity theory: if we measure the complexity of a problem just in terms of the input size n, then reducing the size from n to $n - 1$ in polynomial time yields a polynomial-time algorithm for the problem as we can iterate the reduction [8]. Hence it does not make much sense to study preprocessing for NP-hard problems in the traditional one-dimensional framework. However, the notion of "kernelization", a key concept of Parameterized Complexity provides the means for studying preprocessing, since the impact of preprocessing can measured in terms of the parameter, not the size of the input. When a problem is fixed-parameter tractable then each instance (I, k) can be reduced in polynomial time to an equivalent instance (I', k'), the *problem kernel*, where $k' \leq k$ and the size of I' is bounded by a function of k. The smaller the kernel, the more efficient the fixed-parameter algorithm. For a parameterized problem it is therefore interesting to know whether it admits a *polynomial kernel* or not.

Several optimization problems, such as VERTEX COVER and FEEDBACK VERTEX SET admit polynomial kernels with respect to the standard parameter [4, 3]. However, it turns out that many fixed-parameter tractable problems in the areas of Constraint Satisfaction, Global Constraints, Satisfiability, Nonmonotonic and Bayesian Reasoning do not have polynomial kernels unless the Polynomial Hierarchy collapses to its third level [43]. Such super-polynomial kernel lower bounds can be obtained by means of recent tools [2, 11]. A positive exception is the consistency problem for certain global constraint, parameterized by the number of gaps in the domains of variables, which admits a polynomial kernel [14].

5 Conclusion

Over the last decade, Parameterized Complexity has become an important field of research in Algorithms and Complexity. It allows a more fine-grained complexity analysis than the traditional theory as it allows to take structural aspects of problem instances into account. In this extended abstract we have outlined the basic concepts of Parameterized Complexity and indicated some recent results on the Parameterized Complexity of problems arising in Constraint Satisfaction an Reasoning.

References

1. A. Atserias, A. A. Bulatov, and V. Dalmau. On the power of *k*-consistency. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 279–290. Springer Verlag, 2007.
2. H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels (extended abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 563–574. Springer Verlag, 2008. Full version appeared in the J. of Computer and System Sciences.

3. Y. Cao, J. Chen, and Y. Liu. On feedback vertex set new measure and new structures. In H. Kaplan, editor, *Algorithm Theory - SWAT 2010, 12th Scandinavian Symposium and Workshops on Algorithm Theory, Bergen, Norway, June 21-23, 2010. Proceedings*, volume 6139 of *Lecture Notes in Computer Science*, pages 93–104. Springer Verlag, 2010.
4. J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40–42):3736–3756, 2010.
5. J. Chen and J. Meng. On parameterized intractability: Hardness and completeness. *The Computer Journal*, 51(1):39–59, 2008.
6. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer Verlag, New York, 1999.
7. P. E. Dunne. Computational properties of argument systems satisfying graph-theoretic constraints. *Artificial Intelligence*, 171(10-15):701–729, 2007.
8. M. R. Fellows. The lost continent of polynomial time: Preprocessing and kernelization. In *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings*, volume 4169 of *Lecture Notes in Computer Science*, pages 276–277. Springer Verlag, 2006.
9. J. K. Fichte and S. Szeider. Backdoors to tractable answer-set programming. In T. Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pages 863–868. AAAI Press/IJCAI, 2011.
10. J. Flum and M. Grohe. *Parameterized Complexity Theory*, volume XIV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer Verlag, Berlin, 2006.
11. L. Fortnow and R. Santhanam. Infeasibility of instance compression and succinct PCPs for NP. In C. Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 133–142. ACM, 2008. Full version appeared in the J. of Computer and System Sciences.
12. E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.
13. E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. of the ACM*, 32(4):755–761, 1985.
14. S. Gaspers and S. Szeider. Kernels for global constraints. In T. Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pages 540–545. AAAI Press/IJCAI, 2011.
15. S. Gaspers and S. Szeider. The parameterized complexity of local consistency. in J. H.-M. Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 302–316, Springer Verlag, 2011.
16. P. Giannopoulos, C. Knauer, and S. Whitesides. Parameterized complexity of geometric problems. *The Computer Journal*, 51(3):372–384, 2008.
17. C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.
18. G. Gottlob, R. Pichler, and F. Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 2006.
19. G. Gottlob, R. Pichler, and F. Wei. Abduction with bounded treewidth: From theoretical tractability to practically efficient computation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1541–1546. AAAI Press, 2008.
20. J. Gramm, A. Nickelsen, and T. Tantau. Fixed-parameter algorithms in phylogenetics. *The Computer Journal*, 51(1):79–101, 2008.
21. G. Gutin and A. Yeo. Constraint satisfaction problems parameterized above or below tight bounds: A survey. Technical Report 1108.4803v1, arXiv, 2011.
22. P. Hlinený, S. Oum, D. Seese, and G. Gottlob. Width parameters beyond tree-width and their applications. *The Computer Journal*, 51(3):326–362, 2008.
23. R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. of Computer and System Sciences*, 63(4):512–530, 2001.
24. E. J. Kim, S. Ordyniak, and S. Szeider. Algorithms and complexity results for persuasive argumentation. *Artificial Intelligence*, 175:1722–1736, 2011.
25. A. A. Krokhin and D. Marx. On the hardness of losing weight. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Track A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 662–673. Springer Verlag, 2008.
26. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
27. M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *J. Algorithms*, 31(2):335–354, 1999.

28. M. Mahajan, V. Raman, and S. Sikdar. Parameterizing MAX SNP problems above guaranteed values. In *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings*, volume 4169 of *Lecture Notes in Computer Science*, pages 38–49. Springer Verlag, 2006.
29. U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
30. R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford, 2006.
31. N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Proceedings of SAT 2004 (Seventh International Conference on Theory and Applications of Satisfiability Testing, 10–13 May, 2004, Vancouver, BC, Canada)*, pages 96–103, 2004.
32. N. Nishimura, P. Ragde, and S. Szeider. Solving #SAT using vertex covers. *Acta Informatica*, 44(7-8):509–523, 2007.
33. S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. In K. Lodaya and M. Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 84–95. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
34. S. Ordyniak and S. Szeider. Augmenting tractable fragments of abstract argumentation. In T. Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pages 1033–1038. AAAI Press/IJCAI, 2011.
35. R. Pichler, S. Rümmele, S. Szeider, and S. Woltran. Tractable answer-set programming with weight constraints: Bounded treewidth is not enough. In F. Lin, U. Sattler, and M. Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010.
36. I. Razgon and B. O'Sullivan. Almost 2-SAT is fixed parameter tractable. *J. of Computer and System Sciences*, 75(8):435–450, 2009.
37. M. Samer and S. Szeider. Backdoor sets of quantified Boolean formulas. *Journal of Automated Reasoning*, 42(1):77–97, 2009.
38. M. Samer and S. Szeider. Tractable cases of the extended global cardinality constraint. *Constraints*, 16(1):1–24, 2011.
39. M. Samer and S. Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.
40. M. Samer and S. Szeider. Constraint satisfaction with bounded treewidth revisited. *J. of Computer and System Sciences*, 76(2):103–114, 2010.
41. C. Sloper and J. A. Telle. An overview of techniques for designing parameterized algorithms. *The Computer Journal*, 51(1):122–136, 2008.
42. S. Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005. Reprinted as Chapter 4 of the book SAT 2005 - Satisfiability Research in the Year 2005, edited by E. Giunchiglia and T. Walsh, Springer Verlag, 2006.
43. S. Szeider. Limits of preprocessing. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence, AAAI 2011*, pages 93–98. AAAI Press, Menlo Park, California, 2011.
44. S. Szeider. The parameterized complexity of k-flip local search for SAT and MAX SAT. *Discrete Optim.*, 8(1):139–145, 2011.
45. R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

Recent Advancements in Nonmonotonic Multi-Context Systems*

Michael Fink

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
fink@kr.tuwien.ac.at

Multi-Context Systems (MCS) evolved from seminal work by John McCarthy on contextual reasoning [22], who proposed to consider contexts as abstract objects and formulas to be asserted wrt. such a context. Taking a slightly different point of view, called 'compose-and-conquer', Fausto Giunchiglia and colleagues started to formalize contextual reasoning considering 'context' as a local theory of the world within a network of relations with other local theories [19]. The resulting MCS framework [18] allows to model the information exchange between individual knowledge bases, termed contexts, via so-called *bridge rules*, i.e., rules that represent specific relations between local theories.

The initial MCS formalism, however, required local theories to be represented homogeneously in a monotonic logic, which soon deemed a too restrictive setting. Further developments [20, 24, 6] surpassed these limitations allowing, for instance, for heterogenous but monotonic MCS [20], or for nonmonotonic MCS over knowledge bases represented in Default Logic [6]. In its most recent form, nonmonotonic MCS by Brewka and Eiter [7] generalize in both aspects, interlinking heterogeneous, possibly nonmonotonic knowledge bases through (nonmonotonic) bridge rules. As they provide a principled means to integrate bodies of knowledge formalized by different groups of people without sharing a 'universal' knowledge representation language, nonmonotonic MCS have become a versatile framework in addressing challenges of modern knowledge representation and reasoning [9].

This talk will give a brief overview of Nonomonotonic Multi-Context Systems, before more recent developments are addressed. Syntactically, an MCS is of a collection of contexts, each consisting of a 'logic', a knowledge base, and a set of bridge rules. The notion of logic used here is an abstract way to specify a context formalism in terms of a set of well-formed knowledge bases, a set of possible belief sets, and a so-called acceptability function which assigns to every knowledge base a set of acceptable belief sets. Semantics is given to an MCS by means of belief states, i.e., a sequence of belief sets, one for each context. Intuitively, such a belief state is considered a 'model' of the system if it is in equilibrium. For this, each belief set must be acceptable for the respective context, given its knowledge base and the bridge rules that 'fire' wrt. the belief state under consideration.

An important aspect for the realization of MCS is the availability of a solver to compute equilibria. In contrast to most traditional KR systems, for many practically relevant scenarios, the evaluation of an MCS has to deal with distributed sources of knowledge. We will sketch the principles of a *distributed evaluation algorithm* [11, 3], computing so-called partial equilibria, which has been developed and implemented [4] at TU Wien. A further enhancement of the formalism and its evaluation algorithm are *relational MCS* [16], i.e., an extension towards variables and aggregates in bridge rules.

An MCS which does not have an equilibrium is inconsistent—an undesirable state of affairs for most application scenarios. The goal of *inconsistency management* techniques for MCS [15, 5] is to provide means, like for instance, diagnoses and explanations, in order to analyze and eventually resolve such situations. Investigations on making MCS more robust towards inconsistency recently lead to an innovative, more general advancement of the formalism: while originally bridge rules can only add information to a context, *managed MCS* [10] allow arbitrary operations to be defined (e.g., deletion or revision operators).

The aim of this research, and other ongoing and future work that will be pointed to during this talk, is to underpin and enhance the MCS formalism as to provide the basis of an efficient platform for (distributed) nonmonotonic problem solving on top of heterogeneous distributed knowledge sources.

* This research is partially supported by Austrian Science Fund (FWF) grant P20841, Vienna Science and Technology Fund (WWTF) grant ICT08-020, and the FP7 ICT Project Ontorule (FP7 231875).

Acknowledgements. The progress of work reported is the result of joint research with Gerhard Brewka (University of Leipzig), Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Lucantonio Ghionna (University of Calabria), Thomas Krennwallner, Peter Schüller, and Antonius Weinzierl.

References

1. Adjiman, P., Chatalic, P., Goasdoué, F., Rousset, M.C., Simon, L.: Distributed Reasoning in a Peer-to-Peer Setting: Application to the Semantic Web. J. Artif. Intell. Res. 25, 269–314 (2006)
2. Analyti, A., Antoniou, G., Damásio, C.V.: A principled framework for modular web rule bases and its semantics. In: Proc. KR 2008, pp. 390–400. AAAI Press (2008)
3. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Decomposition of Distributed Nonmonotonic Multi-Context Systems. In: Janhunen and Niemelä (eds.) JELIA. LNCS, vol. 6341, pp. 24–37. Springer (2010)
4. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The DMCS Solver for Distributed Nonmonotonic Multi-Context Systems. In: Janhunen and Niemelä (eds.) JELIA. LNCS, vol. 6341, pp. 352–355. Springer (2010)
5. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The mcs-ie System for Explaining Inconsistency in Multi-Context Systems. In: Janhunen and Niemelä (eds.) JELIA. LNCS, vol. 6341, pp. 356–359. Springer (2010)
6. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: Proc. IJCAI-07, pp. 268–273. (2007)
7. Brewka, G., Eiter, T.: Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In: Proc. AAAI-2007, pp. 385–390. AAAI Press (2007)
8. Brewka, G., Eiter, T.: Argumentation context systems: A framework for abstract group argumentation. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR. LNCS, vol. 5753, pp. 44–57. Springer (2009)
9. Brewka, G., Eiter, T., Fink, M.: Nonmonotonic Multi-Context Systems: A Flexible Approach for Integrating Heterogeneous Knowledge Sources. In: Balduccini, M., Son, T. (eds.) LPNMR. LNCS, vol. 6565, pp. 233–258. Springer (2011)
10. Brewka, G., Eiter, T., Fink, M., Weinzierl, A.: Managed multi-context systems. In: Walsh, T. (ed.) IJCAI. pp. 786–791. IJCAI/AAAI (2011)
11. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed Nonmonotonic Multi-Context Systems. In: Lin, F., Sattler, U. (eds.) KR, pp. 60–70. AAAI Press (2010)
12. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Dynamic Distributed Nonmonotonic Multi-Context Systems. In: Brewka, G., Marek, V., Truszczynski, M. (eds.) Nonmonotonic Reasoning, Essays Celebrating its 30th Anniversary, Studies in Logic, vol. 31. College Publications, London, UK (2011)
13. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321–358 (1995)
14. Eiter, T., Brewka, G., Dao-Tran, M., Fink, M., Ianni, G., Krennwallner, T.: Combining nonmonotonic knowledge bases with external sources. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS. LNCS, vol. 5749, pp. 18–42. Springer (2009)
15. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In: Lin, F., Sattler, U. (eds.) KR, pp. 329–339. AAAI Press (2010)
16. Fink, M., Ghionna, L., Weinzierl, A.: Relational Information Exchange and Aggregation in Multi-Context Systems. In: Delgrande, J.P., Faber, W. (eds.) LPNMR. LNCS, vol. 6645, pp. 120–133. Springer (2011)
17. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Generation Computing 9, 365–385 (1991)
18. Ghidini, C., Giunchiglia, F.: Local models semantics, or contextual reasoning = locality + compatibility. Artif. Intell. 127(2), 221–259 (2001)
19. Giunchiglia, F.: Contextual reasoning. Epistemologia XVI, 345–364 (1993)
20. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: How we can do without modal logics. Artificial Intelligence 65(1), 29–70 (1994)
21. Hirayama, K., Yokoo, M.: The distributed breakout algorithms. Artif. Intell. 161(1–2), 89–115 (Jan 2005)
22. McCarthy, J.: Generality in artificial intelligence. Communications of the ACM 30(12), 1030–1035 (1987)
23. Reiter, R.: A Logic for Default Reasoning. Artif. Intell. 13(1–2), 81–132 (1980)
24. Roelofsen, F., Serafini, L.: Minimal and absent information in contexts. In: Proc. IJCAI-05. (2005)
25. Roelofsen, F., Serafini, L., Cimatti, A.: Many Hands Make Light Work: Localized Satisfiability for Multi-Context Systems. In: de Mántaras, R.L., Saitta, L. (eds.) ECAI, pp. 58–62. IOS Press (2004)
26. Serafini, L., Borgida, A., Tamilin, A.: Aspects of distributed and modular ontology reasoning. In: Proc. IJCAI-05, pp. 570–575. AAAI Press (2005)
27. Serafini, L., Tamilin, A.: Drago: Distributed reasoning architecture for the semantic web. In: ESWC, pp. 361–376. Springer (2005)

A Constraint Logic Programming Approach for Computing Ordinal Conditional Functions

Christoph Beierle[1], Gabriele Kern-Isberner[2], Karl Södler[1]

[1]Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2]Dept. of Computer Science, TU Dortmund, 44221 Dortmund, Germany

Abstract. In order to give appropriate semantics to qualitative conditionals of the form *if A then normally B*, ordinal conditional functions (OCFs) ranking the possible worlds according to their degree of plausibility can be used. An OCF accepting all conditionals of a knowledge base R can be characterized as the solution of a constraint satisfaction problem. We present a high-level, declarative approach using constraint logic programming techniques for solving this constraint satisfaction problem. In particular, the approach developed here supports the generation of all minimal solutions; these minimal solutions are of special interest as they provide a basis for model-based inference from R.

1 Introduction

In knowledge representation, rules play a prominent role. Default rules of the form *If A then normally B* are being investigated in nonmonotonic reasoning, and various semantical approaches have been proposed for such rules. Since it is not possible to assign a simple Boolean truth value to such default rules, a semantical approach is to define when a rational agent accepts such a rule. We could say that an agent accepts the rule *Birds normally fly* if she considers a world with a flying bird to be less surprising than a world with a nonflying bird. At the same time, the agent can also accept the rule *Penguin birds normally do not fly*; this is the case if she considers a world with a nonflying penguin bird to be less surprising than a world with a flying penguin bird.

The informal notions just used can be made precise by formalizing the underlying concepts like default rules, epistemic state of an agent, and the acceptance relation between epistemic states and default rules. In the following, we deal with qualitative default rules and a corresponding semantics modelling the epistemic state of an agent. While a full epistemic state could compare possible worlds according to their possibility, their probability, their degree of plausibility, etc. (cf. [18, 9, 10]), we will use ordinal conditional functions (OCFs), which are also called ranking functions [18]. To each possible world ω, an OCF κ assigns a natural number $\kappa(\omega)$ indicating its degree of surprise: The higher $\kappa(\omega)$, the greater is the surprise for observing ω.

In [12, 13] a criterion when a ranking function respects the conditional structure of a set $\mathcal{R}$ of conditionals is defined, leading to the notion of c-representation for $\mathcal{R}$, and it is argued that ranking functions defined by c-representations are of particular interest for model-based inference. In [3] a system that computes a c-representation for any such $\mathcal{R}$ that is consistent is described, but this c-representation may not be minimal. An algorithm for computing a minimal ranking function is given in [5], but this algorithm fails to find all minimal ranking functions if there is more than one minimal one. In [15] an extension of that algorithm being able to compute all minimal c-representations for $\mathcal{R}$ is presented. The algorithm developed in [15] uses a non-declarative approach and is implemented in an imperative programming language. While the problem of specifying all c-representations for $\mathcal{R}$ is formalized as an abstract, problem-oriented constraint satisfaction problem $CR(\mathcal{R})$ in [2], no solving method is given there.

In this paper, we present a high-level, declarative approach using constraint logic programming techniques for solving the constraint satisfaction problem $CR(\mathcal{R})$ for any consistent $\mathcal{R}$. In particular, the approach developed here supports the generation of all minimal solutions; these minimal solutions are of special interest as they provide a preferred basis for model-based inference from $\mathcal{R}$.

The rest of this paper is organized as follows: After recalling the formal background of conditional logics as it is given in [1] and as far as it is needed here (Section 2), we elaborate the birds-penguins scenario

The research reported here was partially supported by the Deutsche Forschungsgemeinschaft – DFG (grants BE 1700/7-2 and KE 1413/2-2).

sketched above as an illustration for a conditional knowledge base and its semantics in Section 3. The definition of the constraint satisfaction problem $CR(\mathcal{R})$ and its solution set denoting all c-representations for $\mathcal{R}$ is given in Sec. 4. In Section 5, a declarative, high-level CLP program solving $CR(\mathcal{R})$ is developed, observing the objective of being as close as possible to $CR(\mathcal{R})$, and its realization in Prolog is described in detail; in Section 6, it is evaluated with respect to a series of some first example applications. Section 7 concludes the paper and points out further work.

2 Background

We start with a propositional language $\mathcal{L}$, generated by a finite set Σ of atoms $a, b, c, \ldots$. The formulas of $\mathcal{L}$ will be denoted by uppercase Roman letters $A, B, C, \ldots$. For conciseness of notation, we will omit the logical *and*-connective, writing AB instead of $A \wedge B$, and overlining formulas will indicate negation, i.e. $\overline{A}$ means $\neg A$. Let Ω denote the set of possible worlds over $\mathcal{L}$; Ω will be taken here simply as the set of all propositional interpretations over $\mathcal{L}$ and can be identified with the set of all complete conjunctions over Σ. For $\omega \in \Omega$, $\omega \models A$ means that the propositional formula $A \in \mathcal{L}$ holds in the possible world ω.

By introducing a new binary operator $|$, we obtain the set $(\mathcal{L} \mid \mathcal{L}) = \{(B|A) \mid A, B \in \mathcal{L}\}$ of *conditionals* over $\mathcal{L}$. $(B|A)$ formalizes "*if A then (normally) B*" and establishes a plausible, probable, possible etc connection between the *antecedent* A and the *consequence* B. Here, conditionals are supposed not to be nested, that is, antecedent and consequent of a conditional will be propositional formulas.

A conditional $(B|A)$ is an object of a three-valued nature, partitioning the set of worlds Ω in three parts: those worlds satisfying AB, thus *verifying* the conditional, those worlds satisfying $A\overline{B}$, thus *falsifying* the conditional, and those worlds not fulfilling the premise A and so which the conditional may not be applied to at all. This allows us to represent $(B|A)$ as a *generalized indicator function* going back to [7] (where u stands for *unknown* or *indeterminate*):

$$(B|A)(\omega) = \begin{cases} 1 & \text{if } \omega \models AB \\ 0 & \text{if } \omega \models A\overline{B} \\ u & \text{if } \omega \models \overline{A} \end{cases} \tag{1}$$

To give appropriate semantics to conditionals, they are usually considered within richer structures such as *epistemic states*. Besides certain (logical) knowledge, epistemic states also allow the representation of preferences, beliefs, assumptions of an intelligent agent. Basically, an epistemic state allows one to compare formulas or worlds with respect to plausibility, possibility, necessity, probability, etc.

Well-known qualitative, ordinal approaches to represent epistemic states are Spohn's *ordinal conditional functions, OCFs*, (also called *ranking functions*) [18], and *possibility distributions* [4], assigning degrees of plausibility, or of possibility, respectively, to formulas and possible worlds. In such qualitative frameworks, a conditional $(B|A)$ is valid (or *accepted*), if its confirmation, AB, is more plausible, possible, etc. than its refutation, $A\overline{B}$; a suitable degree of acceptance is calculated from the degrees associated with AB and $A\overline{B}$.

In this paper, we consider Spohn's OCFs [18]. An OCF is a function

$$\kappa : \Omega \to \mathbb{N}$$

expressing degrees of plausibility of propositional formulas where a higher degree denotes "less plausible" or "more suprising". At least one world must be regarded as being normal; therefore, $\kappa(\omega) = 0$ for at least one $\omega \in \Omega$. Each such ranking function can be taken as the representation of a full epistemic state of an agent. Each such κ uniquely extends to a function (also denoted by κ) mapping sentences and rules to $\mathbb{N} \cup \{\infty\}$ and being defined by

$$\kappa(A) = \begin{cases} \min\{\kappa(\omega) \mid \omega \models A\} & \text{if } A \text{ is satisfiable} \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

for sentences $A \in \mathcal{L}$ and by

$$\kappa((B|A)) = \begin{cases} \kappa(AB) - \kappa(A) & \text{if } \kappa(A) \neq \infty \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

for conditionals $(B|A) \in (\mathcal{L} \mid \mathcal{L})$. Note that $\kappa((B|A)) \geqslant 0$ since any ω satisfying AB also satisfies A and therefore $\kappa(AB) \geqslant \kappa(A)$.

The belief of an agent being in epistemic state κ with respect to a default rule $(B|A)$ is determined by the satisfaction relation $\models_{\mathcal{O}}$ defined by:

$$\kappa \models_{\mathcal{O}} (B|A) \quad \text{iff} \quad \kappa(AB) \; < \; \kappa(A\overline{B}) \tag{4}$$

Thus, $(B|A)$ is believed in κ iff the rank of AB (verifying the conditional) is strictly smaller than the rank of $A\overline{B}$ (falsifying the conditional). We say that κ *accepts* the conditional $(B|A)$ iff $\kappa \models_{\mathcal{O}} (B|A)$.

3 Example

In order to illustrate the concepts presented in the previous section we will use a scenario involving a set of some default rules representing common-sense knowledge.

Example 1. Suppose we have the propositional atoms f - *flying*, b - *birds*, p - *penguins*, w - *winged* animals, k - *kiwis*.

Let the set $\mathcal{R}$ consist of the following conditionals:

$\mathcal{R}$	$r_1:$	$(f\|b)$	*birds fly*
	$r_2:$	$(b\|p)$	*penguins are birds*
	$r_3:$	$(\overline{f}\|p)$	*penguins do not fly*
	$r_4:$	$(w\|b)$	*birds have wings*
	$r_5:$	$(b\|k)$	*kiwis are birds*

Figure 1 shows a ranking function κ that accepts all conditionals given in $\mathcal{R}$. Thus, for any $i \in \{1,2,3,4,5\}$ it holds that $\kappa \models_{\mathcal{O}} R_i$.

ω	$\kappa(\omega)$	ω	$\kappa(\omega)$	ω	$\kappa(\omega)$	ω	$\kappa(\omega)$
$pbfwk$	2	$p\overline{b}fwk$	5	$\overline{p}bfwk$	0	$\overline{p}\overline{b}fwk$	1
$pbfw\overline{k}$	2	$p\overline{b}fw\overline{k}$	4	$\overline{p}bfw\overline{k}$	0	$\overline{p}\overline{b}fw\overline{k}$	0
$pbf\overline{w}k$	3	$p\overline{b}f\overline{w}k$	5	$\overline{p}bf\overline{w}k$	1	$\overline{p}\overline{b}f\overline{w}k$	1
$pbf\overline{w}\overline{k}$	3	$p\overline{b}f\overline{w}\overline{k}$	4	$\overline{p}bf\overline{w}\overline{k}$	1	$\overline{p}\overline{b}f\overline{w}\overline{k}$	0
$pb\overline{f}wk$	1	$p\overline{b}\overline{f}wk$	3	$\overline{p}b\overline{f}wk$	1	$\overline{p}\overline{b}\overline{f}wk$	1
$pb\overline{f}w\overline{k}$	1	$p\overline{b}\overline{f}w\overline{k}$	2	$\overline{p}b\overline{f}w\overline{k}$	1	$\overline{p}\overline{b}\overline{f}w\overline{k}$	0
$pb\overline{f}\overline{w}k$	2	$p\overline{b}\overline{f}\overline{w}k$	3	$\overline{p}b\overline{f}\overline{w}k$	2	$\overline{p}\overline{b}\overline{f}\overline{w}k$	1
$pb\overline{f}\overline{w}\overline{k}$	2	$p\overline{b}\overline{f}\overline{w}\overline{k}$	2	$\overline{p}b\overline{f}\overline{w}\overline{k}$	2	$\overline{p}\overline{b}\overline{f}\overline{w}\overline{k}$	0

Fig. 1. Ranking function κ accepting the rule set $\mathcal{R}$ given in Example 1

For the conditional $(f|p)$ (*"Do penguins fly?"*) that is not contained in $\mathcal{R}$, we get $\kappa(pf) = 2$ and $\kappa(p\overline{f}) = 1$ and therefore

$$\kappa \not\models_{\mathcal{O}} (f|p)$$

so that the conditional $(f|p)$ is not accepted by κ. This is in accordance with the behaviour of a rational agent believing $\mathcal{R}$ since the knowledge base $\mathcal{R}$ used for building up κ explicitly contains the opposite rule $(\overline{f}|p)$.

On the other hand, for the conditional $(w|k)$ (*"Do kiwis have wings?"*) that is also not contained in $\mathcal{R}$, we get $\kappa(kw) = 0$ and $\kappa(k\overline{w}) = 1$ and therefore

$$\kappa \models_{\mathcal{O}} (w|k)$$

i.e., the conditional $(w|k)$ is accepted by κ. Thus, from their superclass *birds*, kiwis inherit the property of having wings.

4 Specification of Ranking Functions as Solutions of a Constraint Satisfaction Problem

Given a set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of conditionals, a ranking function κ that accepts every R_i repesents an epistemic state of an agent accepting $\mathcal{R}$. If there is no κ that accepts every R_i then $\mathcal{R}$ is *inconsistent*. For the rest of this paper, we assume that $\mathcal{R}$ is consistent.

For any consistent $\mathcal{R}$ there may be many different κ accepting $\mathcal{R}$, each representing a complete set of beliefs with respect to every possible formula A and every conditional $(B|A)$. Thus, every such κ inductively completes the knowledge given by $\mathcal{R}$, and it is a vital question whether some κ' is to be preferred to some other κ'', or whether there is a unique "best" κ. Different ways of determining a ranking function are given by *system Z* [9, 10] or its more sophisticated extension *system Z** [9], see also [6]; for an approach using rational world rankings see [19]. For quantitative knowledge bases of the form $\mathcal{R}_x = \{(B_1|A_1)[x_1], \ldots, (B_n|A_n)[x_n]\}$ with probability values x_i and with models being probability distributions P satisfying a probabilistic conditional $(B_i|A_i)[x_i]$ iff $P(B_i|A_i) = x_i$, a unique model can be choosen by employing the principle of maximum entropy [16, 17, 11]; the maximum entropy model is a best model in the sense that it is the most unbiased one among all models satisfying $\mathcal{R}_x$.

Using the maximum entropy idea, in [13] a generalization of system Z* is suggested. Based on an algebraic treatment of conditionals, the notion of *conditional indifference* of κ with respect to $\mathcal{R}$ is defined and the following criterion for conditional indifference is given: An OCF κ is indifferent with respect to $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ iff $\kappa(A_i) < \infty$ for all $i \in \{1, \ldots, n\}$ and there are rational numbers $\kappa_0, \kappa_i^+, \kappa_i^- \in \mathbb{Q},\ 1 \leqslant i \leqslant n$, such that for all $\omega \in \Omega$,

$$\kappa(\omega) = \kappa_0 + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i B_i}} \kappa_i^+ + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B_i}}} \kappa_i^-. \tag{5}$$

When starting with an epistemic state of complete ignorance (i.e., each world ω has rank 0), for each rule $(B_i|A_i)$ the values κ_i^+, κ_i^- determine how the rank of each satisfying world and of each falsifying world, respectively, should be changed:

- If the world ω verifies the conditional $(B_i|A_i)$, – i.e., $\omega \models A_i B_i$ –, then κ_i^+ is used in the summation to obtain the value $\kappa(\omega)$.
- Likewise, if ω falsifies the conditional $(B_i|A_i)$, – i.e., $\omega \models A_i\overline{B_i}$ –, then κ_i^- is used in the summation instead.
- If the conditional $(B_i|A_i)$ is not applicable in ω, – i.e., $\omega \models \overline{A_i}$ –, then this conditional does not influence the value $\kappa(\omega)$.

κ_0 is a normalization constant ensuring that there is a smallest world rank 0. Employing the postulate that the ranks of a satisfying world should not be changed and requiring that changing the rank of a falsifying world may not result in an increase of the world's plausibility leads to the concept of a *c-representation* [13, 12]:

Definition 1. *Let $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. Any ranking function κ satisfying the conditional indifference condition (5) and $\kappa_i^+ = 0$, $\kappa_i^- \geqslant 0$ (and thus also $\kappa_0 = 0$ since $\mathcal{R}$ is assumed to be consistent) as well as*

$$\kappa(A_i B_i) < \kappa(A_i \overline{B_i}) \tag{6}$$

for all $i \in \{1, \ldots, n\}$ is called a (special) c-representation *of $\mathcal{R}$.*

Note that for $i \in \{1, \ldots, n\}$, condition (6) expresses that κ accepts the conditional $R_i = (B_i|A_i) \in \mathcal{R}$ (cf. the definition of the satisfaction relation in (4)) and that this also implies $\kappa(A_i) < \infty$.

Thus, finding a c-representation for $\mathcal{R}$ amounts to choosing appropriate values $\kappa_1^-, \ldots, \kappa_n^-$. In [2] this situation is formulated as a constraint satisfaction problem $CR(\mathcal{R})$ whose solutions are vectors of the form $(\kappa_1^-, \ldots, \kappa_n^-)$ determining c-representations of $\mathcal{R}$. The development of $CR(\mathcal{R})$ exploits (2) and (5) to reformulate (6) and requires that the κ_i^- are natural numbers (and not just rational numbers). In the following, we set $\min(\emptyset) = \infty$.

Definition 2. ***[CR(R)]*** *Let $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. The constraint satisfaction problem for c-representations of $\mathcal{R}$, denoted by $CR(\mathcal{R})$, is given by the conjunction of the constraints*

$$\kappa_i^- \geqslant 0 \tag{7}$$

$$\kappa_i^- > \min_{\omega \models A_i B_i} \sum_{\substack{j \neq i \\ \omega \models A_j \overline{B_j}}} \kappa_j^- - \min_{\omega \models A_i \overline{B}_i} \sum_{\substack{j \neq i \\ \omega \models A_j \overline{B_j}}} \kappa_j^- \tag{8}$$

for all $i \in \{1, \ldots, n\}$.

A solution of $CR(\mathcal{R})$ is an n-tupel $(\kappa_1^-, \ldots, \kappa_n^-)$ of natural numbers, and with $Sol_{CR}(\mathcal{R})$ we denote the set of all solutions of $CR(\mathcal{R})$.

Proposition 1. *For $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ let $(\kappa_1^-, \ldots, \kappa_n^-) \in Sol_{CR}(\mathcal{R})$. Then the function κ defined by*

$$\kappa(\omega) = \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B}_i}} \kappa_i^- \tag{9}$$

accepts $\mathcal{R}$.

All c-representations built from (7), (8), and (9) provide an excellent basis for model-based inference [13, 12]. However, from the point of view of minimal specificity (see e.g. [4]), those c-representations with minimal κ_i^- yielding minimal degrees of implausibility are most interesting.

While different orderings on $Sol_{CR}(\mathcal{R})$ can be defined, leading to different minimality notions, in the following we will focus on the ordering on $Sol_{CR}(\mathcal{R})$ induced by taking the sum of the κ_i^-, i.e.

$$(\kappa_1^-, \ldots, \kappa_n^-) \leqslant (\kappa'^-_1, \ldots, \kappa'^-_n) \quad \text{iff} \quad \sum_{1 \leqslant i \leqslant n} \kappa_i^- \leqslant \sum_{1 \leqslant i \leqslant n} \kappa'^-_i. \tag{10}$$

As we are interested in minimal κ_i^--vectors, an important question is whether there is always a unique minimal solution. This is not the case; the following example that is also discussed in [15] illustrates that $Sol_{CR}(\mathcal{R})$ may have more than one minimal element.

Example 2. Let $\mathcal{R}_{birds} = \{R_1, R_2, R_3\}$ be the following set of conditionals:

$R_1 : (f\|b)$	*birds fly*
$R_2 : (a\|b)$	*birds are animals*
$R_3 : (a\|fb)$	*flying birds are animals*

From (8) we get

$$\begin{aligned} \kappa_1^- &> 0 \\ \kappa_2^- &> 0 - min\{\kappa_1^-, \kappa_3^-\} \\ \kappa_3^- &> 0 - \kappa_2^- \end{aligned}$$

and since $\kappa_i^- \geqslant 0$ according to (7), the two vectors

$$\begin{aligned} sol_1 &= (\kappa_1^-, \kappa_2^-, \kappa_3^-) = (1, 1, 0) \\ sol_2 &= (\kappa_1^-, \kappa_2^-, \kappa_3^-) = (1, 0, 1) \end{aligned}$$

are two different solutions of $CR(\mathcal{R}_{birds})$ with $\sum_{1 \leqslant i \leqslant n} \kappa_i^- = 2$ that are both minimal in $Sol_{CR}(\mathcal{R}_{birds})$ with respect to $\leqslant$.

5 A Declarative CLP Program for $CR(\mathcal{R})$

In this section, we will develop a CLP program `GenOCF` solving $CR(\mathcal{R})$. Our main objective to obtain a declarative program that is as close as possible to the abstract formulation of $CR(\mathcal{R})$ while exploiting the concepts of constraint logic programming. We will employ finite domain constraints, and from (7) we immediately get a lower bound for κ_i^-. Considering that we are interested mainly in minimal solutions, due to (8) we can safely restrict ourselves to n as an upper bound for κ_i^-, yielding

$$0 \leqslant \kappa_i^- \leqslant n \tag{11}$$

for all $i \in \{1, \ldots, n\}$ with n being the number of conditionals in $\mathcal{R}$.

5.1 Input Format and Preliminaries

Since we want to focus on the constraint solving part, we do not consider reading and parsing a knowledge base $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. Instead, we assume that $\mathcal{R}$ is already given as a Prolog code file providing the following predicates `variables/1`, `conditional/3` and `indices/1`:

`variables([`a_1`,...,`a_m`])`	% list of atoms in Σ
`conditional(`i`,`$\langle A_i \rangle$`,`$\langle B_i \rangle$`)`	% representation of ith conditional $(B_i\|A_i)$
`indices([1,...,`n`])`	% list of indices $\{1, \ldots, n\}$

If $\Sigma = \{a_1, \ldots, a_m\}$ is the set of atoms, we assume a fixed ordering $a_1 < a_2 < \ldots < a_m$ on Σ given by the predicate `variables([`a_1`,...,`a_m`])`.

In the representation of a conditional, a propositional formula A, constituting the antecedent or the consequence of the conditional, is represented by $\langle A \rangle$ where $\langle A \rangle$ is a Prolog list `[`$\langle D_1 \rangle$`,...,`$\langle D_l \rangle$`]`. Each $\langle D_i \rangle$ represents a conjunction of literals such that $D_1 \vee \ldots \vee D_l$ is a disjunctive normal form of A.

Each $\langle D \rangle$, representing a conjunction of literals, is a Prolog list `[`b_1`,...,`b_m`]` of fixed length m where m is the number of atoms in Σ and with $b_k \in \{0, 1, \_\}$. Such a list `[`b_1`,...,`b_m`]` represents the conjunctions of atoms obtained from $\dot{a}_1 \wedge \dot{a}_2 \wedge \ldots \wedge \dot{a}_m$ by eliminating all occurrences of $\top$, where

$$\dot{a}_k = \begin{cases} a_k & \text{if } b_k = 1 \\ \overline{a_k} & \text{if } b_k = 0 \\ \top & \text{if } b_k = \_ \end{cases}$$

Example 3. The internal representation of the knowledge base presented in Example 1 is shown in Figure 2.

```
variables([p,b,f,w,k]).

%               p b f w k     p b f w k
conditional(1,[[_,1,_,_,_]],[[_,_,1,_,_]]).   % (f | b)  birds fly
conditional(2,[[1,_,_,_,_]],[[_,1,_,_,_]]).   % (b | p)  penguins are birds
conditional(3,[[1,_,_,_,_]],[[_,_,0,_,_]]).   % (-f | p) penguins do not fly
conditional(4,[[_,1,_,_,_]],[[_,_,_,1,_]]).   % (w | b)  birds habe wings
conditional(5,[[_,_,_,_,1]],[[_,1,_,_,_]]).   % (b | k)  kiwis are birds

indices([1,2,3,4,5]).
```

Fig. 2. Internal representation of the knowledge base from Example 1

As further preliminaries, using `conditional/3` and `indices/1`, we have implemented the predicates `verifying_worlds/2`, `falsifying_worlds/2`, and `falsify/2`, realising the evaluation of the indicator function (1) given in Sec. 2:

`verifying_worlds(`i`, `Ws`)`	% Ws list of worlds verifying ith conditional
`falsifying_worlds(`i`, `Ws`)`	% Ws list of worlds falsifying ith conditional
`falsify(`i`,`W`)`	% world W falsifies ith conditional

where worlds are represented as complete conjunctions of literals over Σ, using the representation described above.

Using these predicates, in the following subsections we will present the complete source code of the constraint logic program `GenOCF` solving $CR(\mathcal{R})$.

5.2 Generation of Constraints

The particular program code given here uses the SICStus Prolog system[1] and its clp(fd) library implementing constraint logic programming over finite domains [14].

The main predicate `kappa/2` expecting a knowledge base `KB` of conditionals and yielding a vector `K` of κ_i^- values as specified by (8) is presented in Fig. 3.

```
kappa(KB, K) :-          % K is kappa vector of c-representation for KB
   consult(KB),
   indices(Is),          % get list of indices [1,2,...,N]
   length(Is, N),        % N number of conditionals in KB
   length(K, N),         % generate K = [Kappa_1,...,Kappa_N] of free var.
   domain(K, 0, N),      % 0 <= kappa_I <= N  for all I according to (11)
   constrain_K(Is, K),   % generate constraints according to (8)
   labeling([], K).      % generate solution
```

Fig. 3. Main predicate `kappa/2`

After reading in the knowledge base and getting the list of indices, a list `K` of free constraint variables, one for each conditional, is generated. In the two subsequent subgoals, the constraints corresponding to the formulas (11) and (8) are generated, constraining the elements of `K` accordingly. Finally, `labeling([], K)` yields a list of κ_i^- values. Upon backtracking, this will enumerate all possible solutions with an upper bound of n as in (11) for each κ_i^-. Later on, we will demonstrate how to modify `kappa/2` in order to take minimality into account (Sec. 5.3).

How the subgoal `constrain_K(Is, K)` in `kappa/2` generates a constraint for each index $i \in \{1, \ldots, n\}$ according to (8) is defined in Fig. 4.

```
constrain_K([],_).                      % generate constraints for
constrain_K([I|Is],K) :-                % all kappa_I as in (8)
   constrain_Ki(I,K), constrain_K(Is,K).

constrain_Ki(I,K) :-           % generate constraint for kappa_I as in (8)
   verifying_worlds(I, VWorlds),     % all worlds verifying I-th conditional
   falsifying_worlds(I, FWorlds),    % all worlds falsifying I-th conditional
   list_of_sums(I, K, VWorlds, VS),  % VS list of sums for verifying worlds
   list_of_sums(I, K, FWorlds, FS),  % FS list of sums for falsifying worlds
   minimum(Vmin, VS),                % Vmin minium for verifying worlds
   minimum(Fmin, FS),                % Fmin minium for falsifying worlds
   element(I, K, Ki),                % Ki constraint variable for kappa_I
   Ki #> Vmin - Fmin.                % constraint for kappa_I as in (8)
```

Fig. 4. Constraining the vector `K` representing $\kappa_1^-, \ldots, \kappa_n^-$ as in (8)

Given an index `I`, `constrain_Ki(I,K)` determines all worlds verifying and falsifying the `I`-th conditional; over these two sets of worlds the two $\min$ expressions in (8) are defined. Two lists `VS` and `FS` of sums corresponding exactly to the first and the second sum, repectively, in (8) are generated (how this is done is defined in Fig. 5 and will be explained below). With the constraint variables `Vmin` and `Fmin` denoting the minimum of these two lists, the constraint

```
Ki #> Vmin - Fmin
```

[1] http://www.sics.se/isl/sicstuswww/site/index.html

given in the last line of Fig. 4 reflects precisely the restriction on κ_i^- given by (8).

For an index `I`, a kappa vector `K`, and a list of worlds `Ws`, the goal `list_of_sums(I, K, Ws, Ss)` (cf. Fig. 5) yields a list `Ss` of sums such that for each world `W` in `Ws`, there is a sum `S` in `Ss` that is generated by `sum_kappa_j(Js, I, K, W, S)` where `Js` is the list of indices $\{1, \ldots, n\}$. In the goal `sum_kappa_j(Js, I, K, W, S)`, `S` corresponds exactly to the respective sum expression in (8), i.e., it is the sum of all `Kj` such that `J` $\neq$ `I` and `W` falsifies the `j`-th conditional.

```
% list_of_sums(I, K, Ws, Ss)  generates list of sums as in (8):
%     I   index from 1,...,N
%     K   kappa vector
%     Ws  list of worlds
%     Ss  list of sums:
%         for each world W in Ws there is S in Ss s.t.
%            S is sum of all kappa_J with
%            J \= I and W falsifies J-th conditional

list_of_sums(_, _, [], []).
list_of_sums(I, K, [W|Ws], [S|Ss]) :-
   indices(Js),
   sum_kappa_j(Js, I, K, W, S),
   list_of_sums(I, K, Ws, Ss).



% sum_kappa_j(Js, I, K, W, S)  generates a sum as in (8):
%     Js  list of indices [1,...,N]
%     I   index from 1,...,N
%     K   kappa vector
%     W   world
%     S   sum of all kappa_J s.t.
%            J \= I and W falsifies J-th conditional

sum_kappa_j([], _, _, _, 0).
sum_kappa_j([J|Js], I, K, W, S) :-
    sum_kappa_j(Js, I, K, W, S1),
    element(J, K, Kj),
    ((J \= I, falsify(J, W)) -> S #= S1 + Kj; S #= S1).
```

Fig. 5. Generating list of sums of κ_i^- as in (8)

Example 4. Suppose that `kb_birds.pl` is a file containing the conditionals of the knowledge base $\mathcal{R}_{birds}$ given in Ex. 2. Then the first five solutions generated by the program given in Figures 3 – 5 are:

```
| ?- kappa('kb_birds.pl', K).
K = [1,0,1] ? ;
K = [1,0,2] ? ;
K = [1,0,3] ? ;
K = [1,1,0] ? ;
K = [1,1,1] ?
```

Note that the first and the fourth solution are the minimal solutions.

Example 5. If `kb_penguins.pl` is a file containing the conditionals of the knowledge base $\mathcal{R}$ given in Ex. 1, the first six solutions generated by `kappa/2` are:

```
| ?- kappa('kb_penguins.pl', K).
K = [1,2,2,1,1] ? ;
K = [1,2,2,1,2] ? ;
K = [1,2,2,1,3] ? ;
K = [1,2,2,1,4] ? ;
K = [1,2,2,1,5] ? ;
K = [1,2,2,2,1] ?
```

5.3 Generation of Minimal Solutions

The enumeration predicate `labeling/2` of SICStus Prolog allows for an option that minimizes the value of a cost variable. Since we are aiming at minimizing the sum of all κ_i^-, the constraint `sum(K, #=, S)` introduces such a cost variable `S`. Thus, exploiting the SICStus Prolog minimization feature, we can easily modify `kappa/2` to generate a minimal solution: We just have to replace the last subgoal `labeling([], K)` in Fig. 3 by the two subgoals:

```
   sum(K, #=, S),                % introduce constraint variable S
                                 %    for sum of kappa_I
   minimize(labeling([],K), S). % generate single minimal solution
```

With this modification, we obtain a predicate `kappa_min/2` that returns a single minimal solution (and fails on backtracking). Hence calling `?- kappa_min('kb_birds.pl', K).` similar as in Ex. 4 yields the minimal solution `K = [1,0,1]`.

However, as pointed out in Sec. 4, there are good reasons for considering not just a single minimal solution, but all minimal solutions. We can achieve the computation of all minimal solutions by another slight modification of `kappa/2`. This time, the enumeration subgoal `labeling([], K)` in Fig. 3 is preceded by two new subgoals as in `kappa_min_all/2` in Fig. 6.

```
kappa_min_all(KB, K) :-  % K is minimal vector for KB, all solutions
   consult(KB),
   indices(Is),          % get list of indices [1,2,...,N]
   length(Is, N),        % N number of conditionals in KB
   length(K, N),         % generate K = [Kappa_1,...,Kappa_N] of free var.
   domain(K, 0, N),      % 0 <= kappa_I <= N  for all I according to (11)
   constrain_K(Is, K),   % generate constraints according to (8)
   sum(K, #=, S),        % constraint variable S for sum of kappa_I
   min_sum_kappas(K, S), % determine minimal value for S
   labeling([], K).      % generate all minimal solutions

min_sum_kappas(K, Min) :-      % Min is sum of a minimal solution for K
   once((labeling([up],[Min]),
         \+ \+ labeling([],K))).
```

Fig. 6. Predicate `kappa_min_all/2` generating exactly all minimal solutions

The first new subgoal `sum(K, #=, S)` introduces a constraint variable `S` just as in `kappa_min/2`. In the subgoal `min_sum_kappas(K, S)`, this variable `S` is constrained to the sum of a minimal solution as determined by `min_sum_kappas(K, Min)`. These two new subgoals ensure that in the generation caused by the final subgoal `labeling([], K)`, exactly all minimal solutions are enumerated.

Example 6. Continuing Example 4, calling

```
| ?- kappa_min_all('kb_birds.pl', K).
K = [1,0,1] ? ;
K = [1,1,0] ? ;
no
```

yields the two minimal solutions for $\mathcal{R}_{birds}$.

Example 7. For the situation in Ex. 5, `kappa_min_all/2` reveals that there is a unique minimal solution:

```
| ?- kappa_min_all('kb_penguins.pl', K).
K = [1,2,2,1,1] ? ;
no
```

Determining the OCF κ induced by the vector $(\kappa_1^-, \kappa_2^-, \kappa_3^-, \kappa_4^-, \kappa_5^-) = (1, 2, 2, 1, 1)$ according to (9) yields the ranking function given in Fig. 1.

6 Example Applications and First Evaluation

Although the objective in developing `GenOCF` was on being as close as possible to the abstract formulation of the constraint satisfaction problem $CR(\mathcal{R})$, we will present the results of some first example applications we have carried out.

For $n \geqslant 1$, we generated synthetic knowledge bases `kb_synth<`n`>_c<`$2n$–1`>.pl` according to the following schema: Using the variables $\{f\} \cup \{a_1, \ldots, a_n\}$, `kb_synth<`$n$`>_c<`$2n$–1`>.pl` contains the $2 * n - 1$ conditionals given by::

$$\begin{array}{ll} (f|a_i) & \text{if } i \text{ is odd, } i \in \{1, \ldots, n\} \\ (\overline{f}|a_i) & \text{if } i \text{ is even, } i \in \{1, \ldots, n\} \\ (a_i|a_{i+1}) & \text{if } i \in \{1, \ldots, n-1\} \end{array}$$

For instance, `kb_synth4_c7.pl` uses the five variables $\{f, a_1, a_2, a_3, a_4\}$ and contains the seven conditionals:

$$\begin{array}{c} (f|a_1) \\ (\overline{f}|a_2) \\ (f|a_3) \\ (\overline{f}|a_4) \\ (a_1|a_2) \\ (a_2|a_3) \\ (a_3|a_4) \end{array}$$

The basic idea underlying the construction of these synthetic knowledge bases `kb_synth<`n`>_c<`$2n$–1`>.pl` is to establish a kind of subclass relationship between a_{i+1} and a_i for each $i \in \{1, \ldots, n-1\}$ on the one hand, and to state that every a_{i+1} is exceptional to a_i with respect to its behaviour regarding f, again for each $i \in \{1, \ldots, n-1\}$. This sequence of pairwise exceptional elements will force any minimal solution of $CR($`kb_synth<`n`>_c<`$2n$–1`>.pl`$)$ to have at least one κ_i^- value of size greater or equal to n.

From `kb_synth<`n`>_c<`m`>.pl`, the knowledge bases `kb_synth<`n`>_c<`m–j`>.pl` are generated for $j \in \{1, \ldots, m-1\}$ by removing the last j conditionals. For instance, `kb_synth4_c5.pl` is obtained from `kb_synth4_c7.pl` by removing the two conditionals $\{(a_2|a_3), (a_3|a_4)\}$.

Figure 7 shows the time needed by `GenOCF` for computing all minimal solutions for various knowledge bases. The execution time is given in seconds where the value 0 stands for any value less than 0.5 seconds. Measurements were taken for the following environment: SICStus 4.0.8 (x86-linux-glibc2.3), Intel Core 2 Duo E6850 3.00GHz. While the number of variables determines the set of possible worlds, the number of conditionals induces the number of contraints. The values in the table in Fig. 7 give some indication on the influence of both values, the number of variables and the number of conditionals in a knowledge base. For instance, comparing the knowledge base `kb_synth7_c10.pl`, having 8 variables and 10 conditionals, to the knowledge base `kb_synth8_c10.pl`, having 9 variables and also 10 conditionals, we see an increase of the computation time by a factor 2.3. Increasing the number of conditionals, leads to no time increase from `kb_synth7_c10.pl` to `kb_synth7_c11.pl`, and to a time increase factor of about 1.6 when moving from `kb_synth8_c10.pl` to `kb_synth8_c11.pl`, while for moving from `kb_synth8_c10.pl` to `kb_synth9_c10.pl` and `kb_synth10_c10.pl`, we get time increase factors of 3.3 and 11.0, respectively.

Of course, these knowledge bases are by no means representative, and further evaluation is needed. In particular, investigating the complexity depending on the number of variables and conditionals and determining an upper bound for worst-case complexity has still to be done. Furthermore, while the code for `GenOCF` given above uses SICStus Prolog, we also have a variant of `GenOCF` for the SWI Prolog system[2] [20]. In our further investigations, we want to evaluate `GenOCF` also using SWI Prolog, to elaborate the changes required and the options provided when moving between SICStus and SWI Prolog, and to study whether there are any significant differences in execution that might depend on the two different Prolog systems and their options.

7 Conclusions and Further Work

While for a set of probabilistic conditionals $(B_i|A_i)[x_i]$ the principle of maximum entropy yields a unique model, for a set $\mathcal{R}$ of qualitative default rules $(B_i|A_i)$ there may be several minimal ranking functions. In this paper, we developed a CLP approach for solving $CR(\mathcal{R})$, realized in the Prolog program `GenOCF`. The solutions of the constraint satisfaction problem $CR(\mathcal{R})$ are vectors of natural numbers $\vec{\kappa} = (\kappa_1^-, \ldots, \kappa_n^-)$ that uniquely determine an OCF $\kappa_{\vec{\kappa}}$ accepting all conditionals in $\mathcal{R}$. The program `GenOCF` is also able to generate exactly all minimal solutions of $CR(\mathcal{R})$; the minimal solutions of $CR(\mathcal{R})$ are of special interest for model-based inference.

Among the extentions of the approach described here we are currently working on, is the investigation and evaluation of alternative minimality criteria. Instead of ordering the vectors $\vec{\kappa}$ by the sum of their components, we could define a componentwise order on $Sol_{CR}(\mathcal{R})$ by defining $(\kappa_1^-, \ldots, \kappa_n^-) \preceq (\kappa'^-_1, \ldots, \kappa'^-_n)$ iff $\kappa_i^- \leqslant \kappa'^-_i$ for $i \in \{1, \ldots, n\}$, yielding a partial order $\preceq$ on $Sol_{CR}(\mathcal{R})$.

Still another alternative is to compare the full OCFs $\kappa_{\vec{\kappa}}$ induced by $\vec{\kappa} = (\kappa_1^-, \ldots, \kappa_n^-)$ according to (9), yielding the ordering $\preccurlyeq$ on $Sol_{CR}(\mathcal{R})$ defined by $\kappa_{\vec{\kappa}} \preccurlyeq \kappa_{\vec{\kappa}'}$ iff $\kappa_{\vec{\kappa}}(\omega) \leqslant \kappa_{\vec{\kappa}'}(\omega)$ for all $\omega \in \Omega$.

In general, it is an open problem how to strengthen the requirements defining a c-representation so that a unique solution is guaranteed to exist. The declarative nature of constraint logic programming supports easy constraint modification, enabling the experimentation and practical evaluation of different notions of minimality for $Sol_{CR}(\mathcal{R})$ and of additional requirements that might be imposed on a ranking function. Furthermore, in [8] the framework of default rules concidered here is extended by allowing not only default rules in the knowledge base $\mathcal{R}$, but also strict knowledge, rendering some worlds completely impossibe. This can yield a reduction of the problem's complexity, and it will be interesting to see which effects the incorporation of strict knowledge will have on the CLP approach presented here.

References

1. C. Beierle and G. Kern-Isberner. A verified AsmL implementation of belief revision. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *LNCS*, pages 98–111. Springer, 2008.
2. C. Beierle and G. Kern-Isberner. On the computation of ranking functions for default rules – a challenge for constraint programming. In *Proc. Deklarative Modellierung und effiziente Optimierung mit Constraint-Technologie. Workshop at GI Jahrestagung 2011*, 2011. (to appear).
3. C. Beierle, G. Kern-Isberner, and N. Koch. A high-level implementation of a system for automated reasoning with default rules (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of the 4th International Joint Conference on Automated Reasoning (IJCAR-2008)*, volume 5195 of *LNCS*, pages 147–153. Springer, 2008.
4. S. Benferhat, D. Dubois, and H. Prade. Representing default rules in possibilistic logic. In *Proceedings 3th International Conference on Principles of Knowledge Representation and Reasoning KR'92*, pages 673–684, 1992.
5. R. A. Bourne. *Default reasoning using maximum entropy and variable strength defaults*. PhD thesis, Univ. of London, 1999.
6. R.A. Bourne and S. Parsons. Maximum entropy and variable strength defaults. In *Proceedings Sixteenth International Joint Conference on Artificial Intelligence, IJCAI'99*, pages 50–55, 1999.
7. B. DeFinetti. *Theory of Probability*, volume 1,2. John Wiley & Sons, 1974.

[2] http://www.swi-prolog.org/index.html

knowledge base	variables	conditionals	minimal solutions	time
kb_birds.pl	3	3	[[1,0,1],[1,1,0]]	0
kb_penguins.pl	5	5	[[1,2,2,1,1]]	0
kb_synth7_c1.pl	7	1	[[1]]	0
kb_synth7_c2.pl	7	2	[[1,1]]	0
kb_synth7_c3.pl	7	3	[[1,1,1]]	0
kb_synth7_c4.pl	7	4	[[1,1,1,1]]	0
kb_synth7_c5.pl	7	5	[[1,1,1,1,1]]	0
kb_synth7_c6.pl	7	6	[[1,1,1,1,1,1]]	1
kb_synth7_c7.pl	7	7	[[1,1,1,1,1,1,1]]	0
kb_synth7_c8.pl	7	8	[[1,2,1,1,1,1,1,2]]	1
kb_synth7_c9.pl	7	9	[[1,2,2,1,1,1,1,2,3]]	2
kb_synth7_c10.pl	7	10	[[1,2,2,2,1,1,1,2,3,4]]	3
kb_synth7_c11.pl	7	11	[[1,2,2,2,2,1,1,2,3,4,5]]	3
kb_synth7_c12.pl	7	12	[[1,2,2,2,2,2,1,2,3,4,5,6]]	6
kb_synth7_c13.pl	7	13	[[1,2,2,2,2,2,2,2,3,4,5,6,7]]	8
kb_synth8_c1.pl	8	1	[[1]]	0
kb_synth8_c2.pl	8	2	[[1,1]]	0
kb_synth8_c3.pl	8	3	[[1,1,1]]	1
kb_synth8_c4.pl	8	4	[[1,1,1,1]]	0
kb_synth8_c5.pl	8	5	[[1,1,1,1,1]]	1
kb_synth8_c6.pl	8	6	[[1,1,1,1,1,1]]	1
kb_synth8_c7.pl	8	7	[[1,1,1,1,1,1,1]]	2
kb_synth8_c8.pl	8	8	[[1,1,1,1,1,1,1,1]]	3
kb_synth8_c9.pl	8	9	[[1,2,1,1,1,1,1,1,2]]	4
kb_synth8_c10.pl	8	10	[[1,2,2,1,1,1,1,1,2,3]]	8
kb_synth8_c11.pl	8	11	[[1,2,2,2,1,1,1,1,2,3,4]]	11
kb_synth8_c12.pl	8	12	[[1,2,2,2,2,1,1,1,2,3,4,5]]	17
kb_synth8_c13.pl	8	13	[[1,2,2,2,2,2,1,1,2,3,4,5,6]]	27
kb_synth8_c14.pl	8	14	[[1,2,2,2,2,2,2,1,2,3,4,5,6,7]]	38
kb_synth8_c15.pl	8	15	[[1,2,2,2,2,2,2,2,2,3,4,5,6,7,8]]	60
kb_synth9_c1.pl	9	1	[[1]]	0
kb_synth9_c2.pl	9	2	[[1,1]]	0
kb_synth9_c3.pl	9	3	[[1,1,1]]	0
kb_synth9_c4.pl	9	4	[[1,1,1,1]]	2
kb_synth9_c5.pl	9	5	[[1,1,1,1,1]]	2
kb_synth9_c6.pl	9	6	[[1,1,1,1,1,1]]	4
kb_synth9_c7.pl	9	7	[[1,1,1,1,1,1,1]]	6
kb_synth9_c8.pl	9	8	[[1,1,1,1,1,1,1,1]]	9
kb_synth9_c9.pl	9	9	[[1,1,1,1,1,1,1,1,1]]	14
kb_synth9_c10.pl	9	10	[[1,2,1,1,1,1,1,1,1,2]]	26
kb_synth9_c11.pl	9	11	[[1,2,2,1,1,1,1,1,1,2,3]]	41
kb_synth9_c12.pl	9	12	[[1,2,2,2,1,1,1,1,1,2,3,4]]	61
kb_synth9_c13.pl	9	13	[[1,2,2,2,2,1,1,1,1,2,3,4,5]]	88
kb_synth9_c14.pl	9	14	[[1,2,2,2,2,2,1,1,1,2,3,4,5,6]]	127
kb_synth9_c15.pl	9	15	[[1,2,2,2,2,2,2,1,1,2,3,4,5,6,7]]	173
kb_synth9_c16.pl	9	16	[[1,2,2,2,2,2,2,2,1,2,3,4,5,6,7,8]]	256
kb_synth9_c17.pl	9	17	[[1,2,2,2,2,2,2,2,2,2,3,4,5,6,7,8,9]]	361
kb_synth10_c1.pl	10	1	[[1]]	0
kb_synth10_c2.pl	10	2	[[1,1]]	1
kb_synth10_c3.pl	10	3	[[1,1,1]]	1
kb_synth10_c4.pl	10	4	[[1,1,1,1]]	4
kb_synth10_c5.pl	10	5	[[1,1,1,1,1]]	8
kb_synth10_c6.pl	10	6	[[1,1,1,1,1,1]]	15
kb_synth10_c7.pl	10	7	[[1,1,1,1,1,1,1]]	25
kb_synth10_c8.pl	10	8	[[1,1,1,1,1,1,1,1]]	40
kb_synth10_c9.pl	10	9	[[1,1,1,1,1,1,1,1,1]]	61
kb_synth10_c10.pl	10	10	[[1,1,1,1,1,1,1,1,1,1]]	88
kb_synth10_c11.pl	10	11	[[1,2,1,1,1,1,1,1,1,2]]	155
kb_synth10_c12.pl	10	12	[[1,2,2,1,1,1,1,1,1,2,3]]	232
kb_synth10_c13.pl	10	13	[[1,2,2,2,1,1,1,1,1,2,3,4]]	333
kb_synth10_c14.pl	10	14	[[1,2,2,2,2,1,1,1,1,2,3,4,5]]	465
kb_synth10_c15.pl	10	15	[[1,2,2,2,2,2,1,1,1,2,3,4,5,6]]	644
kb_synth10_c16.pl	10	16	[[1,2,2,2,2,2,2,1,1,2,3,4,5,6,7]]	929
kb_synth10_c17.pl	10	17	[[1,2,2,2,2,2,2,2,1,1,2,3,4,5,6,7,8]]	1.174
kb_synth10_c18.pl	10	18	[[1,2,2,2,2,2,2,2,2,1,2,3,4,5,6,7,8,9]]	1.642
kb_synth10_c19.pl	10	19	[[1,2,2,2,2,2,2,2,2,2,2,3,4,5,6,7,8,9,10]]	2.248

Fig. 7. Execution times of `GenOCF` under SICStus Prolog for various knowledge bases

8. T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artif. Intell.*, 142(1):53–89, 2002.
9. M. Goldszmidt, P. Morris, and J. Pearl. A maximum entropy approach to nonmonotonic reasoning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):220–232, 1993.
10. M. Goldszmidt and J. Pearl. Qualitative probabilities for default reasoning, belief revision, and causal modeling. *Artificial Intelligence*, 84:57–112, 1996.
11. G. Kern-Isberner. Characterizing the principle of minimum cross-entropy within a conditional-logical framework. *Artificial Intelligence*, 98:169–208, 1998.
12. G. Kern-Isberner. *Conditionals in nonmonotonic reasoning and belief revision*. Springer, Lecture Notes in Artificial Intelligence LNAI 2087, 2001.
13. G. Kern-Isberner. Handling conditionals adequately in uncertain reasoning and belief revision. *Journal of Applied Non-Classical Logics*, 12(2):215–237, 2002.
14. M.Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, (PLILP'97)*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
15. C. Müller. Implementierung von Default-Regeln durch optimale konditionale Rangfunktionen. Abschlussarbeit Bachelor of Science in Informatik, FernUniversität in Hagen, 2004.
16. J.B. Paris. *The uncertain reasoner's companion – A mathematical perspective*. Cambridge University Press, 1994.
17. J.B. Paris and A. Vencovska. In defence of the maximum entropy inference process. *International Journal of Approximate Reasoning*, 17(1):77–103, 1997.
18. W. Spohn. Ordinal conditional functions: a dynamic theory of epistemic states. In W.L. Harper and B. Skyrms, editors, *Causation in Decision, Belief Change, and Statistics, II*, pages 105–134. Kluwer Academic Publishers, 1988.
19. E. Weydert. System JZ - How to build a canonical ranking model of a default knowledge base. In *Proceedings KR'98*. Morgan Kaufmann, 1998.
20. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010. (to appear in *Theory and Practice of Logic Programming*).

Implementing Equational Constraints in a Functional Language

Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|mh|bjp|fre}@informatik.uni-kiel.de

Abstract. KiCS2 is a new system to compile functional logic programs of the source language Curry into purely functional Haskell programs. The implementation is based on the idea to represent the search space as a data structure and logic variables as operations that generate their values. This has the advantage that one can apply various, and in particular, complete search strategies to compute solutions. However, the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. To overcome this drawback, we propose new techniques to implement equational constraints in this framework. In particular, we show how unification modulo function evaluation and functional patterns can be added without sacrificing the efficiency of the kernel implementation.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [5,17] for recent surveys). In particular, they provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination has led to new design patterns [2,6] and better abstractions for application programming, but it also gave rise to new implementation challenges.

Previous implementations of functional logic languages can be classified into three categories:

1. designing new abstract machines appropriately supporting these operational features and implementing them in some (typically, imperative) language, like C [24] or Java [7,20],
2. compilation into logic languages like Prolog and reusing the existing backtracking implementation for non-deterministic search as well as logic variables and unification for computing with partial information [1,23], or
3. compilation into non-strict functional languages like Haskell and reusing the implementation of lazy evaluation and higher-order functions [13,14].

The latter approach requires the implementation of non-deterministic computations in a deterministic language but has the advantage that the explicit handling of non-determinism allows for various search strategies like depth-first, breadth-first, parallel, or iterative deepening instead of committing to a fixed (incomplete) strategy like backtracking [13].

In this paper we consider KiCS2 [12], a new system that compiles functional logic programs of the source language Curry [21] into purely functional Haskell programs. We have shown in [12] that this implementation can compete with or outperform other existing implementations of Curry. KiCS2 is based on the idea to represent the search space, i.e., all non-deterministic results of a computation, as a data structure that can be traversed by operations implementing various strategies. Furthermore, logic variables are replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the logic variable. It has been shown [4] that computing with logic variables by narrowing [27,30] and computing with generators by rewriting are equivalent, i.e., compute the same values. Although this implementation technique is correct [9], the generation of all values for logic variables might be inefficient for applications that exploit constraints on partial values. For instance, in Prolog the equality constraint "`X=c(a)`" is solved by instantiating the variable `X` to `c(a)`, but the equality constraint "`X=Y`" is solved by binding `X` to `Y` without enumerating any values for `X` or `Y`. Therefore, we propose in this paper new techniques to implement equational constraints in the framework of KiCS2 (note that, in contrast to Prolog,

unification is performed modulo function evaluation). Furthermore, we also show how functional patterns [3], i.e., patterns containing evaluable operations for more powerful pattern matching than in logic or functional languages, can be implemented in this framework. We show that both extensions lead to efficiency improvements without sacrificing the efficiency of the kernel implementation.

In the next section, we review the source language Curry and the features considered in this paper. Section 3 sketches the implementation scheme of KiCS2. Sections 4 and 5 discuss the extensions to implement unification modulo functional evaluation and functional patterns, respectively. Benchmarks demonstrating the usefulness of this scheme are presented in Sect. 6 before we conclude in Sect. 7.

2 Curry Programs

The syntax of the functional logic language Curry [21] is close to Haskell [26], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. Hence, an operation is defined by conditional rewrite rules of the form:

$$f\ t_1 \ldots t_n \ \texttt{|}\ c\ \texttt{=}\ e \quad \texttt{where}\ vs\ \texttt{free} \tag{1}$$

where the *condition* c is optional and vs is the list of variables occurring in c or e but not in the *left-hand side* $f\ t_1 \ldots t_n$.

In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. Such operations are also called *non-deterministic*. For instance, Curry offers a *choice* operation that is predefined by the following rules:

```
x ? _ = x
_ ? y = y
```

Thus, we can define a non-deterministic operation `aBool` by

```
aBool = True ? False
```

so that the expression "`aBool`" has two values: `True` and `False`.

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operations

```
not True  = False
not False = True

xor True  x = not x
xor False x = x

xorSelf x = xor x x
```

and the expression "`xorSelf aBool`". If we interpret this program as a term rewriting system, we could have the reduction

```
xorSelf aBool  →  xor aBool aBool  →  xor True aBool
               →  xor True False   →  not False     →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, González-Moreno et al. [16] proposed the rewriting logic CRWL as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [22] , where values of the arguments of an operation are determined before the operation is evaluated. This can be enforced in a lazy strategy by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False` consistently.

The condition c in rule (1) typically is a conjunction of *equational constraints* of the form $e_1 =:= e_2$. Such a constraint is satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. For instance, if the symbol "++" denotes the usual list concatenation operation, we can define an operation `last` that computes the last element `e` of a non-empty list `xs` as follows:

```
last xs | ys++[e] =:= xs  = e   where ys, e free
```

Like in Haskell, most rules defining functions are *constructor-based* [25], i.e., in (1) $t_1, \ldots, t_n$ consist of variables and/or data constructor symbols only. However, Curry also allows *functional patterns* [3], i.e., t_i might additionally contain calls to defined operations. For instance, we can also define the last element of a list by:

```
last' (xs++[e]) = e
```

Here, the functional pattern `(xs++[e])` states that `(last' t)` is reducible to `e` provided that the argument `t` can be matched against some value of `(xs++[e])` where `xs` and `e` are free variables. By instantiating `xs` to arbitrary lists, the value of `(xs++[e])` is any list having `e` as its last element. Functional patterns are a powerful feature to express arbitrary selections in term structures. For instance, they support a straightforward processing of XML data with incompletely specified or evolving formats [18].

3 The Compilation Scheme of KiCS2

To understand the extensions described in the subsequent sections, we sketch the translation of Curry programs into Haskell programs as performed by KiCS2. More details about this translation scheme can be found in [10,12].

As mentioned in the introduction, the KiCS2 implementation is based on the explicit representation of non-deterministic results in a data structure. This is achieved by extending each data type of the source program by constructors to represent a choice between two values and a failure, respectively. For instance, the data type for Boolean values defined in a Curry program by

```
data Bool = False | True
```

is translated into the Haskell data type[1]

```
data Bool = False | True | Choice ID Bool Bool | Fail
```

The first argument of type `ID` of each `Choice` constructor is used to implement the call-time choice semantics discussed in Sect. 2. Since the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`, we have to ensure that both duplicates, which later evaluate to a non-deterministic choice between two values, yield either `True` or `False`. This is obtained by assigning a unique identifier (of type `ID`) to each `Choice`. The difficulty is to get a unique identifier on demand, i.e., when some operation evaluates to a `Choice`. Since we want to compile into *purely* functional programs (in order to enable powerful program optimizations), we cannot use unsafe features with side effects to generate such identifiers. Hence, we pass a (conceptually infinite) set of identifiers, also called *identifier supply*, to each operation so that a `Choice` can pick its unique identifier from this set. For this purpose, we assume a type `IDSupply`, representing an infinite set of identifiers, with operations

```
initSupply  :: IO IDSupply
thisID      :: IDSupply → ID
leftSupply  :: IDSupply → IDSupply
rightSupply :: IDSupply → IDSupply
```

The operation `initSupply` creates such a set (at the beginning of an execution), the operation `thisID` takes some identifier from this set, and `leftSupply` and `rightSupply` split this set into two disjoint subsets without the identifier obtained by `thisID`. There are different implementations available [8] (see below for a simple implementation) and our system is parametric over concrete implementations of `IDSupply`.

When translating Curry to Haskell, KiCS2 adds to each operation an additional argument of type `IDSupply`. For instance, the operation `aBool` defined in Sect. 2 is translated into:

[1] Actually, our compiler performs some renamings to avoid conflicts with predefined Haskell entities and introduces type classes to resolve overloaded symbols like `Choice` and `Fail`.

```
aBool :: IDSupply → Bool
aBool s = Choice (thisID s) True False
```

Similarly, the operation

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

so that the set `s` is split into a set `(leftSupply s)` containing identifiers for the evaluation of `aBool` and a set `(rightSupply s)` containing identifiers for the evaluation of the operation `xorSelf`.

Since all data types are extended by additional constructors, we must also extend the definition of operations performing pattern matching.[2] For instance, consider the definition of polymorphic lists

```
data List a = Nil | Cons a (List a)
```

and an operation to extract the first element of a non-empty list:

```
head :: List a → a
head (Cons x xs) = x
```

The type definition is then extended as follows:

```
data List a = Nil | Cons a (List a) | Choice ID (List a) (List a) | Fail
```

The operation `head` is extended by an identifier supply and further matching rules:

```
head :: List a → IDSupply → a
head (Cons x xs)      s = x
head (Choice i x1 x2) s = Choice i (head x1 s) (head x2 s)
head _                s = Fail
```

The second rule transforms a non-deterministic argument into a non-deterministic result and the final rule returns `Fail` in all other cases, i.e., if `head` is applied to the empty list as well as if the matching argument is already a failed computation (failure propagation).

To show a concrete example, we use the following implementation of `IDSupply` based on unbounded integers:

```
type IDSupply = Integer
initSupply    = return 1
thisID      n = n
leftSupply  n = 2 * n
rightSupply n = 2 * n + 1
```

If we apply the same transformation to the rules defining `xor` and evaluate the main expression `(main 1)`, we obtain the result

```
Choice 2 (Choice 2 False True) (Choice 2 True False)
```

Thus, the result is non-deterministic and contains three choices, whereby all of them have the same identifier. To extract all values from such a `Choice` structure, we have to traverse it and compute all possible choices but consider the choice identifiers to make consistent (left/right) decisions. Thus, if we select the left branch as the value of the outermost `Choice`, we also have to select the left branch in the selected argument `(Choice 2 False True)` so that `False` is the only value possible for this branch. Similarly, if we select the right branch as the value of the outermost `Choice`, we also have to select the right branch in its selected argument `(Choice 2 True False)`, which again yields `False` as the only possible value. In consequence, the unintended value `True` is not produced.

The requirement to make consistent decisions can be implemented by storing the decisions already made for some choices during the traversal. For this purpose, we introduce the type

```
data Decision = NoDecision | ChooseLeft | ChooseRight
```

[2] To obtain a simple compilation scheme, KiCS2 transforms source programs into uniform programs [12] where pattern matching is restricted to a single argument. This is always possible by introducing auxiliary operations.

where `NoDecision` represents the fact that the value of a choice has not been decided yet. Furthermore, we assume operations to lookup the current decision for a given identifier or change it (depending on the implementation of `IDSupply`, KiCS2 supports several implementations based on memory cells or finite maps):

```
lookupDecision :: ID  → IO Decision
setDecision    :: ID  → Decision  → IO ()
```

Now we can print all values contained in a choice structure in a depth-first manner by the following I/O operation:[3]

```
printValsDFS :: a  → IO ()

printValsDFS Fail             = return ()

printValsDFS (Choice i x1 x2) = lookupDecision i >>= follow
  where
    follow ChooseLeft  = printValsDFS x1
    follow ChooseRight = printValsDFS x2
    follow NoDecision  = do newDecision ChooseLeft  x1
                            newDecision ChooseRight x2

    newDecision d x = do setDecision i d
                         printValsDFS x
                         setDecision i NoDecision

printValsDFS v = print v
```

This operation ignores failures and prints values that are not rooted by a `Choice` constructor. For a `Choice` constructor, it checks whether a decision for this identifier has already been made (note that the initial value for all identifiers is `NoDecision`). If a decision has been made for this choice, it follows this decision. Otherwise, the left alternative is used and this decision is stored. After printing all values w.r.t. this decision, the decision is undone (like in backtracking) and the right alternative is used and stored.

In general, this operation is applied to the normal form of the main expression (where `initSupply` is used to compute an initial identifier supply passed to this expression). The normal form computation is necessary for structured data like lists, so that a failure or choice in some part of the data is moved to the root.

Other search strategies, like breadth-first search, iterative deepening, or parallel search, can be obtained by different implementations of this main operation to print all values. Furthermore, one can also collect all values in a tree-like data structure so that the programmer can implement his own search strategies (this corresponds to encapsulating search [11]). Finally, instead of printing all values, one can easily define operations to print either the first solution only or one by one upon user request. Due to the lazy evaluation strategy of Haskell, such operations can also be applied to infinite choice structures.

To avoid an unnecessary growth of the search space represented by `Choice` constructors, our compiler performs an optimization for deterministic operations. If an operation is defined by non-overlapping rules and does not call, neither directly nor indirectly through other operations, a function defined by overlapping rules, the evaluation of such an operation (like `xor` or `not`) cannot introduce non-deterministic values. Thus, it is not necessary to pass an identifier supply to the operation. In consequence, only the matching rules are extended by additional cases for handling `Choice` and `Fail` so that the generated code is nearly identical to a corresponding functional program. Actually, the benchmarks presented in [12] show that for deterministic operations this implementation outperforms all other Curry implementations, and, for non-deterministic operations, outperforms Prolog-based implementations of Curry and can compete with MCC [24], a Curry implementation that compiles to C.

As mentioned in the introduction, occurrences of logic variables are translated into generators. For instance, the expression "`not x`", where `x` is a logic variable, is translated into "`not (aBool s)`", where `s` is an `IDSupply` provided by the context of the expression. The latter expression is evaluated by reducing the argument `aBool s` to a choice between `True` or `False` followed by applying `not` to this choice. This is similar to a narrowing step on "`not x`" that instantiates the variable `x` to `True` or `False`. Since such gener-

[3] Note that this code has been simplified for readability since the type system of Haskell does not allow this direct definition.

ators are standard non-deterministic operations, they are translated like any other operation and, therefore, do not require any additional run-time support. However, in the presence of equational constraints, there are methods which are more efficient than generating all values. These methods and their implementation are discussed in the next section.

4 Equational Constraints and Unification

As known from logic programming, predicates or constraints are important to restrict the set of intended values in a non-deterministic computation. Apart from user-defined predicates, equational constraints of the form $e_1 =:= e_2$ are the most important kind of constraints. We have already seen a typical application of an equational constraint in the operation `last` in Sect. 2.

Due to the presence of non-terminating operations and infinite data structures, "`=:=`" is interpreted as the *strict equality* on terms [15], i.e., the equation $e_1 =:= e_2$ is satisfied iff e_1 and e_2 are reducible to unifiable constructor terms. In particular, expressions that do not have a value are not equal w.r.t. "`=:=`", e.g., the equational constraint "`head [] =:= head []`" is not satisfiable.[4]

Due to this constructive definition, "`=:=`" can be considered as a binary function defined by the following rules (we only present the rules for the Boolean and list types, where `Success` denotes the only constructor of the type `Success` of constraints):

```
True   =:= True    =  Success
False  =:= False   =  Success

[]     =:= []      =  Success
(x:xs) =:= (y:ys)  =  x =:= y & xs =:= ys

Success & c  =  c
```

If we translate these operations into Haskell by the scheme presented in Sect. 3, the following rules are added to these rules in order to propagate choices and failures:

```
Fail          =:= _             =  Fail
_             =:= Fail          =  Fail
Choice i l r  =:= y             =  Choice i (l =:= y) (r =:= y)
x             =:= Choice i l r  =  Choice i (x =:= l) (x =:= r)
_             =:= _             =  Fail

Fail          & _     =  Fail
Choice i l r  & c     =  Choice i (l & c) (r & c)
_             & _     =  Fail
```

Although this is a correct implementation of equational constraints, it might lead to an unnecessarily large search space when it is applied to generators representing logic variables. For instance, consider the following generator for Boolean lists:

```
aBoolList = [] ? (aBool : aBoolList)
```

This is translated into Haskell as follows:

```
aBoolList :: IDSupply → [Bool]
aBoolList s = Choice (thisID s) [] (aBool (leftSupply s)
                                   : aBoolList (rightSupply s))
```

Now consider the equational constraint "`x =:= [True]`". If the logic variable `x` is replaced by `aBoolList`, the translated expression "`aBoolList s =:= [True]`" creates a search space when evaluating its first argument, although there is no search required since there is only one binding for `x` satisfying the constraint. Furthermore and even worse, unifying two logic variables introduces an infinite search space. For instance, the expression "`xs =:= ys & xs++ys =:= [True]`" results in an infinite search space when the logic variables `xs` and `ys` are replaced by generators.

[4] From now on, we use the standard notation for lists, i.e., `[]` denotes the empty list and `(x:xs)` denotes a list with head element `x` and tail `xs`.

To avoid these problems, we have to implement the idea of the well-known unification principle [28]. Instead of enumerating all values for logic variables occurring in an equational constraint, we *bind* the variables to another variable or term. Since we compile into a purely functional language, the binding cannot be performed by some side effect. Instead, we add binding constraints to the computed results to be processed by a search strategy that extracts values from choice structures.

To implement unification, we have to distinguish free variables from "standard choices" (introduced by overlapping rules) in the target code. For this purpose, we refine the definition of the type `ID` as follows:[5]

```
data ID = ChoiceID Integer | FreeID Integer
```

The new constructor `FreeID` identifies a choice corresponding to a free variable, e.g., the generator for Boolean variables is redefined as

```
aBool s = Choice (FreeID (thisID s)) True False
```

If an operation is applied to a free variable and requires its value, the free variable is transformed into a standard choice. For this purpose, we define a simple operation to perform this transformation:

```
narrow :: ID → ID
narrow (FreeID i) = ChoiceID i
narrow x          = x
```

Furthermore, we use this operation in narrowing steps, i.e., in all rules operating on `Choice` constructors. For instance, in the implementation of the operation `not` we replace the rule

```
not (Choice i x1 x2) s = Choice i (not x1 s) (not x2 s)
```

by the rule

```
not (Choice i x1 x2) s = Choice (narrow i) (not x1 s) (not x2 s)
```

As mentioned above, the consideration of free variables is relevant in equational constraints where *binding constraints* are generated. For this purpose, we introduce a type to represent a binding constraint as a pair of a choice identifier and a decision for this identifier:

```
data Constraint = ID :=: Decision
```

Furthermore, we extend each data type by the possibility to add constraints:

```
data Bool   = ... | Guard [Constraint] Bool
data List a = ... | Guard [Constraint] (List a)
```

A single `Constraint` provides the decision for one constructor. In order to support constraints for structured data, a list of `Constraint`s provides the decision for the outermost constructor and the decisions for all its arguments. Thus, `(Guard` cs v`)` represents a *constrained value*, i.e., the value v is only valid if the constraints cs are consistent with the decisions previously made during search. These binding constraints are created by the equational constraint operation "`=:=`": if a free variable should be bound to a constructor, we make the same decisions as it would be done in the successful branch of the generator. In case of Boolean values, this can be implemented by the following additional rules for "`=:=`":

```
Choice (FreeID i) _ _ =:= True  =  Guard [i :=: ChooseLeft ] Success
Choice (FreeID i) _ _ =:= False =  Guard [i :=: ChooseRight] Success
```

Hence, the binding of a variable to some known value is implemented as a binding constraint for the choice identifier for this variable. However, if we want to bind a variable to another variable, we cannot store a concrete decision. Instead, we store the information that the decisions for both variables, when they are made to extract values, must be identical. For this purpose, we extend the `Decision` type to cover this information:

```
data Decision = ... | BindTo ID
```

Furthermore, we add the rule that an equational constraint between two variables yields a binding for these variables:

```
Choice (FreeID i) _ _ =:= Choice (FreeID j) _ _
  =  Guard [i :=: BindTo j] Success
```

[5] For the sake of simplicity, in the following, we consider the implementation of `IDSupply` to be unbounded integers.

The consistency of constraints is checked when values are extracted from a choice structure, e.g., by the operation `printValsDFS`. For this purpose, we extend the definition of the corresponding search operations by calling a constraint solver for the constraints. For instance, the definition of `printValsDFS` is extended by a rule handling constrained values:

```
...
printValsDFS (Guard cs x) = do consistent <- add cs
                               if consistent then do printValsDFS x
                                                     remove cs
                                             else return ()
...
```

The operation `add` checks the consistency of the constraints `cs` with the decisions made so far and, in case of consistency, stores the decisions made by the constraints. In this case, the constrained value is evaluated before the constraints are removed to allow backtracking. Furthermore, the operations `lookupDecision` and `setDecision` are extended to deal with bindings between two variables, i.e., they follow variable chains in case of `BindTo` constructors.

Finally, with the ability to distinguish free variables (choices with an identifier of the form `(FreeID ...)`) from other values during search, values containing logic variables can also be printed in a specific form rather than enumerating all values, similarly to logic programming systems. For instance, KiCS2 evaluates the application of `head` to an unknown list as follows:

```
Prelude> head xs where xs free
{xs = (_x2:_x3)} _x2
```

Here, free variables are marked by the prefix `_x`.

5 Functional Patterns

A well-known disadvantage of equational constraints is the fact that "`=:=`" is interpreted as strict equality. Thus, if one uses equational constraints to express requirements on arguments, the resulting operations might be too strict. For instance, the equational constraint in the condition defining `last` (see Sect. 2) requires that `ys++[e]` as well as `xs` must be reducible to unifiable terms so that in consequence the input list `xs` is completely evaluated. Hence, if `failed` denotes an operation whose evaluation fails, the evaluation of `last [failed,True]` has no result. On the other hand, the evaluation of `last' [failed,True]` yields the value `True`, i.e., the definition of `last'` is less strict thanks to the use of functional patterns.

As another example for the advantage of the reduced strictness implied by functional patterns, consider an operation that returns the first duplicate element in a list. Using equational constraints, we can define it as follows:

```
fstDup xs | xs =:= ys++[e]++zs & elem e ys =:= True & nub ys =:= ys
          = e    where ys, zs, e free
```

The first equational constraint is used to split the input list `xs` into three sublists. The last equational constraint ensures that the first sublist `ys` does not contain duplicated elements (the library operation `nub` removes all duplicates from a list) and the second equational constraint ensures that the first element after `ys` occurs in `ys`. Although this implementation is concise, it cannot be applied to infinite lists due to the strict interpretation of "`=:=`". This is not the case if we define this operation by a functional pattern:

```
fstDup' (ys++[e]++zs) | elem e ys =:= True & nub ys =:= ys
                      = e
```

Because of the reduced strictness, the logic variable `zs` (matching the tail list after the first duplicate) is never evaluated. This is due to the fact that a functional pattern like `(xs++[e])` abbreviates all values to which it can be evaluated (by narrowing), like `[e]`, `[x1,e]`, `[x1,x2,e]` etc. Conceptually, the rule defining `last'` abbreviates the following (infinite) set of rules:

```
last' [e] = e
last' [x1,e] = e
last' [x1,x2,e] = e
...
```

Obviously, one cannot implement functional patterns by a transformation into an infinite set of rules. Instead, they are implemented by a specific *lazy unification* procedure "=:<=" [3]. For instance, the definition of `last'` is transformed into

```
last' ys | (xs++[e]) =:<= ys  = e   where xs, e free
```

The behavior of "=:<=" is similar to "=:=", except for the case that a variable in the left argument should be bound to some expression: instead of evaluating the expression to some value and binding the variable to the value, the variable is bound to the *unevaluated* expression (see [3] for more details). Due to this slight change, failures or infinite structures in actual arguments do not cause problems in the matching of functional patterns.

The general structure of the implementation of functional patterns in KiCS2 is quite similar to that of equational constraints, with the exception that variables could be also bound to unevaluated expressions. Only if such variables are later accessed, the expressions they are bound to are evaluated. This can be achieved by adding a further alternative to the type of decisions:

```
data Decision = ... | LazyBind [Constraint]
```

The implementation of the lazy unification operation "=:<=" is almost identical to the strict unification operation "=:=" as shown in Sect. 4. The only difference is in the rules where a free variable occurs in the left argument. All these rules are replaced by the single rule

```
Choice (FreeID i) _ _ =:<= x
  = Guard [i :=: LazyBind (lazyBind i x)] Success
```

where the auxiliary operation `lazyBind` implements the demand-driven evaluation of the right argument `x`:

```
lazyBind :: ID → a → [Constraint]
lazyBind i True  = [i :=: ChooseLeft]
lazyBind i False = [i :=: ChooseRight]
```

The use of the additional `LazyBind` constructor allows the argument `x` to be stored in a binding constraint without evaluation (due to the lazy evaluation strategy of the target language Haskell). However, it is evaluated by `lazyBind` when its binding is required by another part of the computation. Similarly to equational constraints, lazy bindings are processed by a solver when values are extracted. In particular, if a variable has more than one lazy binding constraint (which is possible if a functional pattern evaluates to a non-linear term), the corresponding expressions are evaluated and unified according to the semantics of functional patterns [3].

In order to demonstrate the operational behavior of our implementation, we sketch the evaluation of the lazy unification constraint `xs++[e] =:<= [failed,True]` that occurs when the expression `last' [failed,True]` is evaluated (we omit failed branches and some other details; note that logic variables are replaced by generators, i.e., we assume that `xs` is replaced by `aBoolList 2` and `e` is replaced by `aBool 3`):

```
    aBoolList 2 ++ [aBool 3] =:<= [failed, True]
 ⇝ [aBool 4, aBool 3] =:<= [failed, True]
 ⇝ aBool 4 =:<= failed & aBool 3 =:<= True & [] =:<= []
 ⇝ Guard [ 4 :=: LazyBind (lazyBind 4 failed)
         , 3 :=: LazyBind (lazyBind 3 True)] Success
```

If the value of the expression `last' [failed,True]` is later required, the value of the variable `e` (with the identifier `3`) is in turn required. Thus, `(lazyBind 3 True)` is evaluated to `[3 :=: ChooseLeft]` which corresponds to the value `True` of the generator `(aBool 3)`. Note that the variable with identifier `4` does not occur anywhere else, so that the binding `(lazyBind 4 failed)` will never be evaluated, as intended.

6 Benchmarks

In this section we evaluate our implementation of equational constraints and functional patterns by some benchmarks. The benchmarks were executed on a Linux machine running Debian 5.0.7 with an Intel Core

Expression	==	=:=	=:<=
`last (map (inc 0) [1..10000])`	2.91	0.05	0.01
`simplify`	10.30	6.77	7.07
`varInExp`	2.34	0.24	0.21
`fromPeano (half (toPeano 10000))`	26.67	5.95	11.19
`palindrome`	30.86	14.05	20.26
`horseman`	3.24	3.31	n/a
`grep`	1.06	0.10	n/a

Fig. 1. Benchmarks: comparing different representations for equations

Expression	KiCS2	PAKCS	MCC
`last (map (inc 0) [1..10000])`	0.05	0.40	0.01
`simplify`	6.77	0.15	0.00
`varInExp`	0.24	0.89	0.07
`fromPeano (half (toPeano 10000))`	5.95	108.88	3.22
`palindrome`	14.05	32.56	1.07
`horseman`	3.31	8.70	0.42
`grep`	0.10	2.88	0.14

Fig. 2. Benchmarks: strict unification in different Curry implementations

2 Duo (3.0GHz) processor. KiCS2 has been used with the Glasgow Haskell Compiler (GHC 7.0.4, option -O2) as its backend and an efficient `IDSupply` implementation that makes use of `IORef`s. For a comparison with other mature implementations of Curry, we considered PAKCS [19] (version 1.9.2, based on a SICStus-Prolog 4.1.2) and MCC [24] (version 0.9.10). The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark as a mean of three runs. The programs used for the benchmarks, partially taken from [3], are `last` (compute the last element of a list),[6] `simplify` (simplify a symbolic arithmetic expression), `varInExp` (non-deterministically return a variable occuring in a symbolic arithmetic expression), `half` (compute the half of a Peano number using logic variables), `palindrome` (check whether a list is a palindrome), `horseman` (solving an equation relating heads and feet of horses and men based on Peano numbers), and `grep` (string matching based on a non-deterministic specification of regular expressions [5]).

In Sect. 4 we mentioned that equational constraints could also be solved by generators without variable bindings, but this technique might increase the search space due to the possibly superfluous generation of all values. To show the beneficial effects of our implementation of equational constraints with variable bindings, in Fig. 1 we compare the results of using equational constraints (`=:=`) to the results where the Boolean equality operator (==) is used (which does not perform bindings but enumerate all values). As expected, in most cases the creation and traversal of a large search space introduced by == is much slower than our presented approach with variable bindings. In addition, the example `last` shows that the lazy unification operator ("`=:<=`") improves the performance when unifying an expression which has to be evaluted only partially. Using strict unification, all elements of the list are (unnecessarily) evaluated.

In contrast to the Curry implementations PAKCS and MCC, our implementation of strict unification is based on an explicit representation of the search space instead of backtracking and manipulating a global state containing bindings for logic variables. Nevertheless, the benchmarks in Fig. 2, using equational constraints only, show that it can compete with or even outperform the other implementations. The results show that the implementation of unification of MCC performs best. However, in most cases our implementation outperforms the Prolog-based PAKCS implementation, except for some examples. In particular, `simplify` does not perform well due to expensive bindings of free variables to large arithmetic expressions in unsuccessful branches of the search. Further investigation and optimization will hopefully lead to a better performance in such cases.

[6] "`inc x n`" is a naive addition that `n` times increases its argument `x` by 1.

Expression	KiCS2	PAKCS
`last (map (inc 0) [1..10000])`	0.01	0.33
`simplify`	7.07	0.27
`varInExp`	0.21	1.87
`fromPeano (half (toPeano 10000))`	11.19	∞
`palindrome`	20.26	∞

Fig. 3. Benchmarks: functional patterns in different Curry implementations

As MCC does not support functional patterns, the performance of lazy unification is compared with PAKCS only (Fig. 3). Again, our compiler performs well against PAKCS and outperforms it in most cases ("∞" denotes a run time of more than 30 minutes).

7 Conclusions and Related Work

We have presented an implementation of equational constraints and functional patterns in KiCS2, a purely functional implementation of Curry. Our implementation is based on adding binding constraints to computed values and processing them when values are extracted at the top level of a computation. Since we only have added new constructors and pattern matching rules for them in our implementation, no overhead is introduced for programs without equational constraints, i.e., our implementation does not sacrifice the high efficiency of the kernel implementation shown in [12]. However, if these features are used, they usually lead to a comparably efficient execution, as demonstrated by our benchmarks.

Other implementations of equational constraints in functional logic languages use side effects for their implementation. For instance, PAKCS [19] exploits the implementation of logic variables in Prolog, which are implemented on the primitive level by side effects. MCC [24] compiles into C where a specific abstract machine implements the handling of logic variables. We have shown that our implementation is competitive to those. In contrast to those systems, our implementation supports a variety of search strategies, like breadth-first or parallel search, where the avoidance of side effects is important.

For future work it might be interesting to add further constraint structures to our implementation, like real arithmetic or finite domain constraints. This might be possible by extending the kinds of constraints of our implementation and solving them by functional approaches like [29].

References

1. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
2. S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.
3. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.
7. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
8. L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.
9. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.

10. B. Braßel and S. Fischer. From functional logic programs to purely functional programs preserving laziness. In *Pre-Proceedings of the 20th Workshop on Implementation and Application of Functional Languages (IFL 2008)*, 2008.
11. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
12. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
13. B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.
14. B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.
15. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.
16. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
17. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
18. M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.
19. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.
20. M. Hanus and R. Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999(6), 1999.
21. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
22. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
23. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
24. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
25. M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
26. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
27. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
28. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
29. T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, 2009.
30. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

Transfer of Semantics from Argumentation Frameworks to Logic Programming – a Preliminary Report

Monika Adamová and Ján Šefránek

Comenius University, Bratislava, Slovakia,
monika.adamova@gmail.com; sefranek@ii.fmph.uniba.sk

Abstract. There are various interesting semantics' (extensions) designed for argumentation frameworks. They enable to assign a meaning, e.g., to odd-length cycles. Our main motivation is to transfer semantics' proposed by Baroni, Giacomin and Guida for argumentation frameworks with odd-length cycles to logic programs with odd-length cycles through default negation. The developed construction is even stronger. For a given logic program an argumentation framework is defined. The construction enables to transfer each semantics of the resulting argumentation framework to a semantics of the given logic program. Weak points of the construction are discussed and some future continuations of this approach are outlined.

Keywords: argumentation framework; extension; logic program; odd cycle; semantics

1 Introduction

Relations between (extensions of) abstract argumentation frameworks and (semantics of) logic programs were studied since the fundamental paper by Dung [3] and since the times of other seminal paper [10]. We can mention also, e.g., [7, 18, 2, 8, 9, 11–17].

Among typical research problems are, e..g.,

- a characterization of extensions of abstract argumentation framework in terms of answer sets or other semantics' of logic programs,
- a construction of new semantics of logic programs, based or inspired by extensions of argumentation frameworks,
- encoding extensions in answer set programming.

Our main motivation is to transfer semantics' proposed in [5] for argumentation frameworks with odd-length cycles to logic programs with odd-length cycles through default negation. According to our knowledge, only CF2 extensions of [5], were studied from different logic programming points of view, see, e.g., [11, 17]. In [11] an ASP-encoding of (modified) CF2 is presented and in [17] a characterization of CF2 in terms of answer set models is proposed.

Our goal is to propose some new semantics' of logic programs (we are primarily interested in a semantic handling of odd cycles through default negation) via transferring semantics' of argumentation frameworks (AD1, AD2, CF1, CF2). We propose a uniform method, which for a given logic program transfers arbitrary argumentation semantics to a semantics of the logic program. The method enables to define for a given logic program a corresponding argumentation framework. As next step, each semantics of the resulting argumentation framework is transferred to a semantics of the given logic program.

This paper is structured as follows. Basics of SCC-recursive semantics of [5] is sketched after technical preliminaries. Then, in Section 4, the core of the paper, a transfer of argumentation framework semantics' to logic program is described. A special attention is devoted to the problem of odd cycles in the Section 5. A representation of an argumentation framework A by a logic program P is described in Section 6. It is shown that for an arbitrary argumentation semantics holds that extensions of the original argumentation framework A coincide with extensions of the argumentation framework constructed for P using the method of Section 4. Weak points of the construction are discussed in the paper. Some future continuations of this research are outlined in Section 7. Finally, related work is overviewed and main contributions, open problems and future goals are summarized in Conclusions.

2 Preliminaries

Some basic notions of argumentation frameworks and logic programs are introduced in this section.

Argumentation frameworks An *argumentation framework* [3] is a pair $AF = (AR, atatcks)$, where AR is a set (of arguments) and $attacks \subseteq AR \times AR$ is a binary relation. Let be $a, b \in AR$; if $(a, b) \in atatcks$, it is said that a attacks b. We assume below an argumentation framework $AF = (AR, attacks)$.

Let be $S \subseteq AR$. It is said that S is *conflict-free* if for no $a, b \in S$ holds $(a, b) \in attacks$.

A set of arguments $S \subseteq AR$ attacks $a \in AR$ iff there is $b \in S$ s.t. $(b, a) \in attacks$.

A conflict-free set of arguments S is *admissible* in AF iff for each $a \in S$ holds: if there is $b \in AR$ s.t. $(b, a) \in attacks$, then S attacks b, i.e. an admissible set of arguments counterattacks each attack on its members.

Dung defined some semantic characterizations (extensions) of argumentation frameworks as sets of conflict-free and admissible arguments, which satisfy also some other conditions.

A *preferred extension* of AF is a maximal admissible set in AF. A conflict-free $S \subseteq AR$ is a *stable extension* of AF iff S attacks each $a \in AR \setminus S$.

The *characteristic function* F_{AF} of an argumentation framework AF assigns sets of arguments to sets of arguments, where $F_{AF}(S) = \{a \in AR \mid \forall b \in AR \ (b \text{ attacks } a \Rightarrow S \text{ attacks } b\}$.

The *grounded* extension of an argumentation framework AF is the least fixed point of F_{AF} (F_{AF} is monotonic).

A *complete extension* is an admissible set S of arguments s.t. each argument, which is acceptable with respect to S, belongs to S.

We will use a precise notion of a semantics of an argumentation framework. A *semantics* of AF is a mapping σ_*, which assigns a set of extensions to AF. Different indices in the place of * specify different semantics', e.g. preferred semantics, stable semantics etc. A set of extensions assigned by a semantics $\mathcal{S}$ to an argumentation framework AF is denoted by $\mathcal{E}_S(AF)$.

Logic programs Only propositional normal logic programs are considered in this paper. Let $\mathcal{L}$ be a set of atoms. The set of default literals is $not\ \mathcal{L} = \{not\ A \mid A \in \mathcal{L}\}$. A literal is an atom or a default literal. A rule (let us denote it by r) is an expression of the form

$$A \leftarrow A_1, \ldots, A_k, not\ B_1, \ldots, not\ B_m; \text{ where } k \geq 0, m \geq 0 \tag{1}$$

A is called the head of the rule and denoted by $head(r)$.
The set of literals $\{A_1, \ldots, A_k, not\ B_1, \ldots, not\ B_m\}$ is called the body of r and denoted by $body(r)$. $\{A_1, \ldots, A_k\}$, called the positive part of the body, is denoted by $body^+(r)$ and $\{B_1, \ldots, B_m\}$ is denoted by $body^-(r)$. Notice that $body^-(r)$ differs from the negative part $\{not\ B_1, \ldots, not\ B_m\}$ of the body.

A (normal) program is a finite set of rules. We will often use only the term program.

We will specify a transfer of an argumentation semantics to a logic program semantics in terms of sets of atoms derivable in the corresponding logic program. We follow the approach of Dimopoulos and Torres [6] in order to specify a notion of derivation in a normal logic program. The derivation should be dependent on a set of default literals. In the next paragraphs we will adapt some basic definitions from [6].

An *assumption* is a default literal. A set of assumptions Δ is called a *hypothesis*. $\Delta^{\hookrightarrow^P}$ is a set of atoms, dependent on (derivable from) Δ w.r.t. a program (set of rules) P; here is a precise definition:

Let Δ, a hypothesis be given. P_Δ is the set of all rules from P, where elements from Δ are deleted from the bodies of the rules and P^+_Δ is obtained from P_Δ by deleting all rules r with bodies containing assumptions. Then $\Delta^{\hookrightarrow^P} = \{A \in \mathcal{L} \mid P^+_\Delta \models A\}$).

It is said that an atom A is *derived* from Δ using rules of P iff $A \in \Delta^{\hookrightarrow^P}$.

Stable model semantics of logic programs play a background role in our paper, so, we introduce a definition of stable model. An interpretation $S = \Delta \cup \Delta^{\hookrightarrow^P}$ is a stable model of P iff S is total interpretation [6], where an interpretation is understood as a consistent set of literals.

3 SCC-recursive semantics

An analysis of asymmetries in handling of even and odd cycles in argumentation semantics' is presented in [5]. We present only a sketchy view of their approach, for details see [5].

An argumentation framework may be conceived as an oriented graph with arguments as vertices and the attack relation as the set of edges.

Example 1 *Consider* $AF = (\{a, b, c\}, \{(a, b), (b, c), (c, a)\})$. *The graph representation of AF contains an odd-length cycle.*

This example is often presented as a case o three witnesses and the attack relation is interpreted as follows: a *questions reliability of* b, b *questions reliability of* c, c *questions reliability of* a.

Stable semantics does not assign an extension to such argumentation framework. However, there are two stable extensions for the case of four witnesses.

This asymmetry in semantic treatment of odd and even cycles motivated the research and solutions of [5]. The same problem is present in a form also in other "classical" argumentation semantics proposed in [3]. □

A general recursive schema for argumentation semantics is proposed in [5]. Recursive semantics' are defined in a constructive way – an incremental process of adding arguments into an extension is specified.

A symmetric handling of odd and even cycles is based on distinguishing components of graphs.

Definition 1 *Let an argumentation framework* $AF = \langle AR, attacks \rangle$ *be given. A binary relation of path equivalence, denoted by* $PE_{AF} \subseteq (AR \times AR)$, *is defined as follows.*

- $\forall a \in AR, (a, a) \in PE_{AF}$,
- $\forall a \neq b \in AR, (a, b) \in PE_{AF}$ *iff there is a path from* a *to* b *and a path from* b *to* a.

The strongly connected components *of* AF *are the equivalence classes of arguments (vertices) under the relation of path-equivalence. The set of the strongly connected components of* AF *is denoted by* $SCCS_{AF}$.

We now can consider the set of strongly connected components as the set of vertices of a new graph. Consider components C_1 and C_2. Let an argument a be a member of C_1 and b be a member of C_2. If a attacks b (in AF), then (C_1, C_2) is an edge of the graph of strongly connected components (SCC-graphs). It is clear that this graph is an acyclic one.

Notions of parents and ancestors for SCC-graphs are defined in an obvious way. Initial components (components without parents) provide a basis for a construction of an extension. We start at the initial component and proceed via oriented edges to next components. If we construct an extension E and a component C is currently processed, the process consists in a choice of a subset of C, i.e. a choice of $E \cap C$ (according to the given semantics – the semantics specifies how choices depend on choices made in ancestors of C). A base function is assumed, which is applied to argumentation frameworks with exactly one component and it characterizes a particular argumentation semantics.

A notion of SCC-recursive argumentation semantics formalizes the intuitions presented above. SCC-recursive characterization of traditional semantics' is provided. Finally, some new semantics', AD1, AD2, CF1 and CF2, are defined in [5].

AD1 and AD2 extensions preserve the property of admissibility. However, the requirement of maximality is relaxed, so this solution is different as compared to the preferred semantics. An alternative is not to require admissibility of sets of arguments and and insist only on conflict-freeness. Maximal conflict-free sets of arguments are selected as extensions in semantics CF1 and CF2. For details and differences see [5]. ASP-encodings of AD1, AD2, CF1 and CF2 are presented in [1].

4 Transfer of argumentation framework semantics' to logic program

We will build an argumentation framework over the rules of a logic program. Rules will play the role of arguments. An attack relation over such arguments will be introduced. After that some arguments (rules)

are accepted/rejected on the basis of a given argumentation semantics. A corresponding semantics for logic program is introduced as a set of literals derivable from accepted rules (considered as arguments). Note that this method enables a transfer of an arbitrary argumentation semantics to the given logic program.

Definition 2 *Let a program P be given. Then an* argumentation framework over P *is* $AF_P = \langle AR, attacks \rangle$, *where*
$AR = \{r \in P\}$ *and* $attacks = \{(r_1, r_2) \mid A = head(r_1), body^+(r_1) = \emptyset, A \in body^-(r_2)\}$. □

Example 2 *Let be* $P = \{r_1 : a \leftarrow;\ r_2 : b \leftarrow not\ a.\}$. *Then* $attacks = \{(r_1, r_2)\}$ *in* AF_P.
If $P = \{r_1 : a \leftarrow not\ b.\ r_2 : b \leftarrow not\ a.\}$, *then* $attacks = \{(r_1, r_2), (r_2, r_1)\}$. □

Let us discuss the condition that the attacking rules do not contain positive literals in its body. A derivation of the head of a rule r with non-empty $body^+(r)$ from a hypothesis Δ is conditional: it depends on a derivation of positive literals in $body^+(r)$. We constrain the attacking argument in the attack relation to the rules with non-empty $body^+(r)$ – it is recognizable on syntactic level and it is appropriate for the representation of argumentation frameworks in logic programs presented in Section 6.

But this design decision leads to some counterintuitive consequences in a general case. We will return to the problem below, after formal definitions.

We have defined an argumentation framework over the rules of a program P. Let's proceed towards derivations in P, based on an argumentation semantics.

Let a program P be given, AF_P be an argumentation framework over P. Consider a set of rules $R \subseteq P$, where R is a conflict-free set of arguments of AF_P. It is obvious that R could serve as a basis of a reasonable derivation in the corresponding logic program. Only literals which do not occur as negated in the bodies of rules are in the heads of rules.

Notice that extensions of an argumentation framework over a program P are sets of rules. That is expressed by a notion of rules enabled in a program P by an argumentation semantics according to the following definition.

Definition 3 *A set of rules $R \subseteq P$ is* enabled *in a program P by an argumentation semantics S iff $R \in \mathcal{E}_S(AF_P)$. If R satisfies this condition, it is denoted by $Rule\_in_S^P$ (or by a shorthand $Rule\_in$, if a given semantics and a given program are clear from the context).* □

A set of rules R ($Rule\_in_S^P$) is enabled by S according to Definition 3, if R is an S-extension of AF_P. The following definition of a set of atoms consistent with a set of rules is important. It partially prevents some negative consequences of the decision that attacking rules have empty positive part of the body. Inconsistent sets of rules cannot be derived because of checking consistency, see Definition 6.

Definition 4 *Let M be an arbitrary set of atoms and $R \subseteq P$ be an arbitrary subset of a program$P$.*
It is said that M is consistent *with R iff* $\forall A \in M\ \neg\exists r \in R\ A \in body^-(r)$. □

Now, a fundamental task is to point out a way from $Rule\_in_S^P$, rules enabled by an argumentation semantics to a corresponding set of atoms, i.e., to a semantics of the given logic program P. The set is denoted by $In\_AS_{\mathcal{S}}$, see the following definition.

Definition 5 *Let AF_P be an argumentation framework over a program P, $\mathcal{S}$ be an argumentation semantics of AF_P and $Rule\_in_{\mathcal{S}}$ is a set of rules of P enabled by the semantics $\mathcal{S}$.*
Then $In\_AS_{\mathcal{S}}$ is the least set of atoms A satisfying the following condition:
$\exists r \in Rule\_in_S, head(r) = A, \forall b \in body^+(r) : b \in In\_AS_{\mathcal{S}}$. □

Definition 5 specifies how to compute In_AS. First, for each $r \in Rule\_in_S$ s.t. $body^+(r) = \emptyset$ and $head(r) = A$, A is included into In_AS. After that is In_AS iteratively recomputed for all $r \in Rule\_in_S$ with non-empty $body^+(r)$. Notice that this is a process of $T_{Rule\_in_S}$-iteration.

Finally, it is necessary to use consistent $In\_AS_{\mathcal{S}}$ in order to define a sound semantic characterization of the given logic program P. This characterization is called the set of atoms *derived* in P according to semantics $\mathcal{S}$ according to the following definition.

Definition 6 *If $In\_AS_{\mathcal{S}}$ is consistent with $Rule\_in_{\mathcal{S}}$, then it is said that $In\_AS_{\mathcal{S}}$ is the set of atoms* derived *in P according to semantics $\mathcal{S}$.* □

Example 3 *Let a program $P = \{r_1 : a \leftarrow, r_2 : b \leftarrow not\ a, r_3 : c \leftarrow not\ b, r_4 : d \leftarrow not\ c\}$ be given.*

We get $AF_P = (\ r_1, r_2, r_3, r_4\}, \{(r_1, r_2), (r_2, r_3), (r_3, r_4)\})$. Consider only the preferred semantics. The only preferred extension of AF_P is the set of rules $\{r_1, r_3\}$.[1] We get $\{\{r_1, r_3\}\} = \mathcal{E}_S(AF_P)$, where S is the preferred semantics. It means, $\{r_1, r_3\}$ is the only set of rules, enabled by the preferred semantics according to Definition 3.

$In\_AS = \{a, c\}$ according to Definition 5. The set of atoms $\{a, c\}$ is consistent with the set of rules $\{r_1, r_3\}$ according to the Definition 4. Finally, according to Definition 6 is $\{a, c\}$ derived in P according to the preferred semantics.

Notice that this set is the stable model of P. □

Example 4 *Consider now a less straightforward example.*

Let P be $\{r_1 : a \leftarrow not\ b,\ r_2 : b \leftarrow c, not\ d.\}, r_3 : c \leftarrow .\}$, then $attacks = \emptyset$. If S is the preferred semantics, then $\{\{r_1, r_2, r_3\}\} = \mathcal{E}_S(AF_P)$, $P = Rule\_in_{\mathcal{S}}$ is enabled by the preferred semantics.

Further, it holds that $In\_AS_{\mathcal{S}} = \{a, b, c\}$ according to Definition 5. But $In\_AS_{\mathcal{S}}$ is not consistent with $P = Rule\_in_{\mathcal{S}}$, hence no atom is derived in P according to the preferred semantics.

Consistency checks are intended as a guard against hidden attacks, as our example demonstrates. This is why the set $In\_AS_{\mathcal{S}}$ is not derivable in P according to the preferred semantics. Hence, our construction prevent to accept inconsistent sets of atoms as semantic characterizations of logic programs.

On the other hand, $\{r_2, r_3\}$ (may be, also $\{r_1, r_3\}$) could be an intuitive preferred extension of an argumentation framework assigned to P. It means that our construction do not generate all intuitive semantic characterizations of a logic program corresponding to an argumentation semantics. □

Remark 1 *May be, a way out of this bug could be built over subsets of $Rule\_in_{\mathcal{S}}$ and/or of In_AS. Definition 6 can be modified accordingly as follows: Let M be a maximal subset of $In\_AS_{\mathcal{S}}$ and R be a maximal subset of $Rule\_in_{\mathcal{S}}$ s.t. M is consistent with R. Then it is said that M is the set of atoms* derived *in P according to semantics $\mathcal{S}$.*

If we consider Example 4, we get sets $\{a, c\}$ and $\{b, c\}$ as derived atoms corresponding to the preferred extension. However, this is not appropriate for stable semantics. A nice uniform transfer of an argumentation semantics to a logic program semantics would be lost, if a special handling of inconsistency for different argumentation semantics' is specified.

More comments about some possible ways how to fix this bug are included into Section 7. □

We repeat that the given construction of an argumentation framework over a logic program is useful for goals of Section 6. Possibilities of more general constructions aiming at a transfer of an argumentation semantics to a logic program semantics are presented in Section 7.

Derivation of atoms according to Definition 6 coincides with the derivation of derivation in Section 2.

Proposition 1 *Let an argumentation semantics S be given. Let be $R = Rule\_in_S$. A set of atoms derived in P according to the semantics S is $\Delta^{\sim_R}$ for some Δ.*

Proof:
Let be $R = Rule\_in_S$ and In_AS be the corresponding derived set of atoms.

Suppose that $\Delta = \{not\ A \mid \exists r \in R\ A \in body^-(r)\}$. It holds that $A \in \Delta^{\sim^R}$ iff $R^+_\Delta \models A$. Obviously, $R^+_\Delta \models A$ holds iff $A \in In\_AS$. □

An open problem is, how semantics' transferred from argumentation frameworks are related to known semantics of logic programs (stable model semantics, partial stable model semantics, well founded semantics etc.)

Note that stable extensions of AF_P are not in general stable models of P.

[1] It is also a stable, grounded and complete extension.

Example 5 *Consider the program* $P = \{r_1 : a \leftarrow p, not\ b, r_2 : b \leftarrow q, not\ a, r_3 : p \leftarrow\}$.

The stable model of P *is* $\{p, a\}$*, but the stable extension of* AF_P *does not exist, rules* r_1, r_2, r_3 *are mutually conflict-free, but* $In\_AS = \{a, b, p\}$ *is not consistent with* $Rule\_in = \{r_1, r_2, r_3\}$. □

This observation is a consequence of the given design decision concerning the attack relation – attacking rules are only rules with empty positive part of the body.

5 Odd cycles

In this section some examples are presented in order to show that a transfer of an argumentation semantics to a logic program (without a suitable "classic" semantic characterization) enables a reasonable semantic characterization of the program.

Some logic programs without stable models have a clear intuitive meaning. A transfer of argumentation semantics from the corresponding argumentation framework enables to catch a meaning of such programs. Of course, a more detailed analysis is needed, in order to understand the relations of those semantics to partial stable models semantics and well founded semantics (or other semantics' of logic programs).[2]

Example 6 *Remind Example 3. Let* P' *be* $P \cup \{r5 : e \leftarrow not\ e\}$. P' *has no stable model.*

The graph of the argumentation framework $AF_{P'}$ *contains an isolated vertex* $r5$ *which attacks itself. If we transfer preferred and grounded semantics from* $AF_{P'}$ *to logic program* P'*, we obtain a semantic characterization by an intuitive set of rules* $\{r_1, r_3\}$ *and, consequently, of atoms* $\{a, c\}$ *as in Example 3.* □

However, a special interest deserves the problem of odd cycles. In this case a transfer from argumentation semantics' to logic program semantics' provides a new perspective on logic programs.[3]

Example 7 *Consider program* $P_1 = \{r_1 : a \leftarrow not\ b, r_2 : b \leftarrow not\ a\}$ *with an even (negative) cycle and* $P_2 = \{r_1 : a \leftarrow not\ b, r_2 : b \leftarrow not\ c, r_3 : c \leftarrow not\ a\}$ *with an odd (negative) cycle. There is no stable model of* P_2.

Preferred, stable and complete argumentation semantics' assign two extensions to AF_{P_1}*. On the other hand, they assign one (empty) or no extension to* AF_{P_2}.

Recursive semantics' proposed in [5] overcome this asymmetry. Note that AF_P *consists of the only component, the odd cycle* $(r_1, r_2), (r_2, r_3), (r_3, r_1)$*.* $CF1$ *assigns three extensions* $\{\{a\}, \{b\}, \{c\}\}$ *to this framework. Our construction enables to transfer this semantics to the logic program* P_2. □

Consider also other example.

Example 8 *Let be* $P = \{r_1 : a \leftarrow not\ a, r_2 : b \leftarrow not\ a\}$*. The argumentation framework* AF_P *has according to the semantics* $CF2$ *extension* r_2*, consequently* $\{b\}$ *is transferred to* P. □

6 Representation of argumentation framework by logic program

In this section we apply a changed view. An argumentation framework AF is assumed and its representation by a simple logic program P_{AF} is constructed. Then we can construct an argumentation framework $\mathcal{A}$ over the rules of that program using the method of Section 4. Suppose that an argumentation semantics S is applied to the argumentation framework $\mathcal{A}$ over the rules of the program P_{AF}. We will show that an application of transferred argumentation semantics to the logic program P_{AF} produces the same result as the application of the semantics to the original argumentation framework AF.

Definition 7 *Let an argumentation framework* $AF = \langle AR, attacks \rangle$ *be given. We represent* AF *by a logic program* P_{AF} *as follows*

– *for each* $a \in AR$ *there is exactly one rule* $r \in P_{AF}$ *s.t.* $head(r) = \{a\}$

[2] Some results are presented in the literature, see Section 8.

[3] We realize that this is a complex problem and diverse intuitions should be analyzed.

– $body^-(r) = \{b \mid b \in AR, (b,a) \in attacks\}$, $body^+(r) = \emptyset$.

□

A remark: if body of a rule is empty, then the corresponding argument is not attacked in AF.

Example 9 *Let be $AF = (AR, attacks)$, where $AR = \{a,b,c,d,e\}$ and $attacks = \{(a,b),(c,b),(c,d),(d,c),(d,e),(e,e)\}$. P_{AF}, the logic program representing AF is as follows:*

$$r_1 : b \leftarrow not\ a, not\ c$$
$$r_2 : a \leftarrow$$
$$r_3 : c \leftarrow not\ d$$
$$r_4 : d \leftarrow not\ c$$
$$r_5 : e \leftarrow not\ e, not\ d$$

□

Programs representing an argumentation framework look like lists: to each argument in the head of a rule is assigned a list of arguments attacking the argument in the head of the rule.

Notice that there are logic programs, which cannot represent an argumentation framework. On the other hand, if a logic program represents an argumentation framework, it is done in a unique way – there is exactly one argumentation framework represented by the program.

Example 10 $P_1 = \{a \leftarrow not\ b, b \leftarrow not\ a\}$ *is a logic program, which represents the argumentation framework* $AF = \langle\{a,b\},\{(a,b),(b,a)\}\rangle$.

$P_2 = \{a \leftarrow not\ b\}$ *cannot be a representation of any argumentation framework. There is no rule in* P_2 *with* b *in its head (and each argument must be in the head of a rule).*

Theorem 2 *Let AF be an argumentation framework, $AF = (AR, attacks)$, P_{AF} be the logic program representing AF. Let In_AS be a set of atoms, derivable in P_{AF} according to a semantics S.*

Then In_AS is an extension of AF according to the semantics S.

Proof:
For each argument $a \in AR$, there is exactly one rule $r \in P_{AF}$ s.t. $head(r) = a$. A function $\Psi : R \rightarrow AR$, where $R \subseteq P$, assigns to each rule $r \in R$ the argument $a \in AR$, which occurs in the head of r. $\Psi^{-1} : AR \rightarrow R$ is an inverse function which assigns to an argument the rule with the argument in the head.

$In\_AS = \{a \mid \exists r \in Rule\_in, head(r) = a\}$ follows from the fact that $body^+(r) = \emptyset$ for each rule r. Hence, $In\_AS = \Psi(Rule\_in)$.

It follows from the definition that for each $(a,b) \in attacks$ there is a pair $(r_1, r_2) \in attacks_P$, where $AF_P = \langle AR_P, attacks_P\rangle$. Notice that $a \in head(r_1)$ and in $head(r_2)$ is b. (AF_P is a framework over the rules of the program P). If $(a,b) \in attacks$ then $not\ a$ occurs in the body of a rule with b in the head. Similarly, for all $(r_1, r_2) \in attacks_P$ there is $(x,y) \in AF$ s.t. $head(r_1) = x, y \in body^-(r_2)$. Therefore, the only difference between the frameworks AF and AF_P is that the vertices of both frameworks are renamed according to the function Ψ.

Therefore, $In\_AS = \Psi(Rule\_in) = \mathcal{E}_S(AF)$.

7 Future goals

In this section three possible alternative transfers of argumentation semantics' to logic program semantics' are sketched. Only very preliminary remarks are presented.

Canonical program. The first possibility, which we will investigate is as follows. Suppose, that an argumentation framework is given. We can represent the argumentation framework by a logic program P_{AF} defined in Section 6 or by its more limpid, straightforward copy P^{AF} defined below.

Definition 8 *Let $AF = (AR, attacks)$ be an argumentation framework. The logic program P^{AF} assigned to AF is the least set of rules satisfying the conditions:*

- *AR is the set of atoms of P^{AF},*
- *if $(a, b) \in attacks$, then $(a \leftarrow not\ b) \in P^{AF}$,*
- *if $a \in AR$ and neither $(a, b) \in attacks$, nor $(b, a) \in atacks$ for some b, then $(a \leftarrow) \in P^{AF}$.*

□

It can be said, that P^{AF} is the *canonical* logic program w.r.t. AF. An argumentation semantics of AF can be transferred to a semantics of the canonical program in a rather straightforward way (in terms of dependencies on hypotheses). The planned next step is a transfer of those dependencies to arbitrary logic programs (for some argumentation semantics' a similar work is done by [10]).

Hypotheses as arguments. Dung in his seminal paper [3] proposed a representation of a logic program in an argumentation framework. Pairs of the form (Δ, A), where Δ is a hypothesis and $A \in \Delta^{\frown_P}$ are arguments in [3].[4]

While Dung was focused on expressing a logic program as an argumentation framework, our goal is to transfer argumentation semantics "back" to the logic program. An interesting contribution could be a transfer of AD1, AD2, CF1, CF2 and other new semantics specified for AF^P back to P. We will use some notions of [6] in order to present a similar idea how to consider hypotheses as arguments.

Definition 9 ([6]) *A hypothesis Δ attacks another hypothesis Δ' in a program P if there is $A \in \Delta^{\frown_P}$ s.t. $not\ A \in \Delta'$.*

A hypothesis Δ is self-consistent in P, if it does not attack itself □

Definition 10 *Let a program P be given. Let $\mathcal{H}$ be the set of all hypothesis over the language of P.*

Then an associated argumentation framework $AF^P = (AR, attacks)$ is defined as follows. AR is the set of all self-consistent hypotheses of $\mathcal{H}$ and $attacks$ is defined as in Definition 9.

If $E \in \mathcal{E}_S(AF^P)$ for a semantics S, then for each $\Delta \in E$ the set of atoms $\Delta^{\frown_P}$ provides a semantic characterization of P according to S □

Notice that this construction is computationally more demanding – AF^P cannot be constructed by an inspection of the syntactic form of P.

Moreover, it is possible that to an extension E of $\mathcal{E}_S(AF^P)$ is assigned a set of sets of atoms of P. It seems that only maximal (w.r.t set-theoretic inclusion) hypotheses of E should be considered if e.g. preferred semantics is transferred.

If we consider Example 4, which illustrates a counterintuitive properties of AF_P, constructed in Section 4, we get an intuitive solution.

Example 11 *Let P be as in Example 4. Then AR of AF^P, the set of self-consistent hypotheses in P is $\{\emptyset, \{not\ a\}, \{not\ b\}, \{not\ d\}, \{not\ a, not\ d\}\}$*
and $attacks = \{(\{not\ d\}, \{not\ b\}), (not\ a, not\ d\}, \{not\ b\})$.

We get that $E = \{\emptyset, \{not\ a\}, \{not\ d\}, \{not\ a, not\ d\}\}$ is a preferred extension. If only maximal hypotheses are considered, the set of atoms $\{b, c\}$ is the transferred semantic characterization of P. Otherwise, both $\{c\}$ and $\{b, c\}$ correspond to E. □

We have to study the details and consequences of the presented proposal.

[4] But in [4] arguments are hypotheses, too.

Derivation of arguments A bug caused by assumption $body^+(r) = \emptyset$ in Definition 2 can be fixed using the approach of [19]. Basic argumentation structures and basic attacks are assumed. Basic argumentation structures contain also conditional arguments. A kind of unfolding of conditional arguments is possible thanks to derivation rules, which enable to derive (non-basic) argumentation structures. Similarly, other derivation rules enable derivation of attacks between general argumentation structures. This machinery enables to leave out the condition $body^+(r) = \emptyset$ of Definition 2.

8 Related work

This section contains only some sketchy remarks, a more detailed analysis and comparison is planned.

We are familiar with the following types of results: a correspondence of an argumentation semantics and a logic program semantics is described, particularly, a characterization of extensions of abstract argumentation framework in terms of answer sets or other semantics' of logic programs. Encoding extensions of argumentation frameworks in answer set programming is another type of research. Some researchers construct a new semantics of logic programs, inspired by extensions of argumentation frameworks. This goal is close to ours. However, every result about relations between an argumentation semantics and logic program semantics is helpful for our future research.

Some remarks concerning Dung's approach were presented in previous section.

Relations between the "classic" argumentation semantics' and corresponding semantic views on logic programs is studied in [10]. Of course, the problem of odd cycles is not tackled in the paper. Our future goal is a detailed comparison of constructions of [10] and ours.

Argumentation framework is constructed and studied in terms of logic programs in [18]. Arguments are expressed in a logic programming language, conflicts between arguments are decided with the help of priorities on rules.

A theory of argumentation that can deal with contradiction within an argumentation framework was presented in [7]. The results was applied to logic programming semantics. A new semantics of logic programs was proposed. The goal is similar as ours, we will devote an attention to this result.

The correspondence between complete extensions in abstract argumentation and 3-valued stable models in logic programming was studied in [2].

The project "New Methods for Analyzing, Comparing, and Solving Argumentation Problems", see, e.g., [9, 8, 11], is focused on implementations of argumentation frameworks in Answer-Set Programming, but also other fundamental theoretical questions are solved. CF2 semantics is studied, too. An Answer Set Programming Argumentation Reasoning Tool (ASPARTIX) is evolved.

The Mexican group [12–17] contributes to research on relations of logic programing and argumentation frameworks, too. Their attention is devoted to characterizations of argumentation semantics' in terms of logic programming semantics'. Also a characterization of CF2 is provided in terms of answer set models or stratified argumentation semantics, which is based on stratified minimal models of logic programs.

Our main goal, in the context of presented remarks, is to "import" semantics' from argumentation frameworks to logic programs. However, results about relations of both areas are relevant for us.

9 Conclusions

A method for transferring an arbitrary argumentation semantics to a logic program semantics was developed. The method consists in defining an argumentation framework over the rules of a program. Extensions of the argumentation framework are sets of rules. A set of consequences of those rules is an interpretation, which provides the corresponding semantic characterization of the program.

This method allows a semantic characterization of programs with odd-length (negative) cycles. If a simple program is assigned to an argumentation framework, extensions of the original framework and the framework over the rules of that program coincide.

The presented method prevents generation of inconsistent sets of atoms. On the other hand, it does not create sometimes a semantic characterization of the original program, even if there is an intuitive possibility to specify the semantics. Some ways of solving this bug are sketched in the paper.

Open problems, future goals and connections to related work are discussed in previous sections.

Acknowledgements: We are grateful to anonymous referees for valuable comments and proposals. This paper was supported by the grant 1/0689/10 of VEGA.

References

1. Monika Adamová: Representácia abstraktného argumentačného frameworku logickým programom; Master Thesis, Comenius University, 2011
2. Wu, Y., Caminada, M., Gabbay, D.: Complete Extensions in Argumentation Coincide with Three-Valued Stable Models in Logic Programming. Studia Logica 93(2-3):383-403 (2009)
3. Phan Minh Dung *On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games* Artificial Intelligence 77, pages 321-357, 1995.
4. Dung, P.M.: An argumentation semantics for logic programming with explicit negation. ICLP'93 Proceedings of the tenth international conference on logic programming. MIT Press Cambridge, MA, USA 1993
5. Baroni, P., Giacomin, M., Guida, G.: SCC- recursiveness: a general schema for argumentation semantics. Artificial Intelligence, 168 (1-2), 2005, 162-210
6. Yannis Dimopoulos, Alberto Torres: Graph theoretical structures in logic programs and default theories, Theoretical Computer Science 170, pages 209-244, 1996.
7. Jakobovits, H., Vermeir, D.: Contradiction in Argumentation Frameworks. Proceedings of the IPMU conference, 1996, 821–826.
8. Uwe Egly, Sarah Alice Gaggl, Stefan Woltran *Answer Set Programming Encodings for Argumentation Frameworks* DBAI Technical Report, DBAI-TR-2008-62, 2008.
9. Eggly, U., Gaggl, A., Woltran, S.: ASPARTIX: Implementing Argumentation Frameworks Using Answer-Set Programming. Proceedings of the 24th International Conference on Logic Programming (ICLP 2008), pages 734-738. Springer LNCS 5366, 2008.
10. A. Bondarenko, P.M. Dung, R.A. Kowalski, F. Toni *An abstract, argumentation-theoretic approach to default reasoning. Artif. Intell. 93: 63-101 (1997)*
11. Sarah Alice Gaggl, Stefan Woltran *cf2 Semantics Revisited* Frontiers in Artificial Intelligence and Applications, pages 243-254. IOS Press, 2010.
12. J. L. Carballido, J. C. Nieves, and M. Osorio.: Inferring Preferred Extensions by Pstable Semantics. Iberoamerican Journal of Artificial Intelligence (Inteligencia Artificial) ISSN: 1137-3601, 13(41):3853, 2009 (doi: 10.4114/ia.v13i41.1029).
13. J. C. Nieves, M. Osorio, and U. Cortés. Preferred Extensions as Stable Models. Theory and Practice of Logic Programming, 8(4):527543, July 2008.
14. J. C. Nieves, M. Osorio, and C. Zepeda. Expressing Extension-Based Semantics based on Stratified Minimal Models. In H. Ono, M. Kanazawa, and R. de Queiroz, editors, Proceedings of WoLLIC 2009, Tokyo, Japan, volume 5514 of FoLLI-LNAI subseries, pages 305319. Springer Verlag, 2009.
15. M. Osorio, A. Marin-George, and J. C. Nieves. Computing the Stratified Minimal Models Semantic. In LANMR'09, pages 157-171, 2009.
16. Juan Carlos Nieves and Ignasi Gomez-Sebastia: Extension-Based Argumentation Semantics via Logic Programming Semantics with Negation as Failure. Proceedings of the Latin-American Workshop on Non-Monotonic Reasoning, CEUR Workshop Proceedings vol 533, ISSN 1613-0073, pages 31-45, Apizaco, Mexico, November 5-6, 2009.
17. Osorio, M., Nieves, J.C., Gmez-Sebastia, I.: CF2-extensions as Answer-set Models. Proceedings of the COMMA2010 Conference. Pages 391-402.
18. H. Prakken, G. Sartor: Argument-based logic programming with defeasible priorities. Journal of Applied Nonclassical Logics 7: 25-75 (1997), special issue on 'Handling inconsistency in knowledge systems'.
19. Ján Šefránek and Alexander Šimko: Warranted derivation of preferred answer sets, http://kedrigern.dcs.fmph.uniba.sk/reports/, TR-2011-027, Comenius University, Faculty of Mathematics, Physics, ans Informatics, 2011. Accepted for WLP 2011.

Translating Nondeterministic Functional Language based on Attribute Grammars into Java

Masanobu Umeda[1], Ryoto Naruse[2], Hiroaki Sone[2], and Keiichi Katamine[1]

[1] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka 820-8502, Japan
umerin@ci.kyutech.ac.jp
[2] NaU Data Institute Inc., 680-41 Kawazu, Iizuka 820-8502, Japan

Abstract. Knowledge-based systems are suitable for realizing advanced functions that require domain-specific expert knowledge, while knowledge representation languages and their supporting environments are essential for realizing such systems. Although Prolog is useful and effective in realizing such a supporting environment, the language interoperability with other implementation languages, such as Java, is often an important issue in practical application development. This paper describes the techniques for translating a knowledge representation language that is a nondeterministic functional language based on attribute grammars into Java. The translation is based on binarization and the techniques proposed for Prolog to Java translation although the semantics are different from those of Prolog. A continuation unit is introduced to handle continuation efficiently, while the variable and register management on backtracking is simplified by using the single and unidirectional assignment features of variables. An experimental translator written in the language itself successfully generates Java code, while experimental results show that the generated code is over 25 times faster than that of Prolog Cafe for nondeterministic programs, and over 2 times faster for deterministic programs. The generated code is also over 2 times faster than B-Prolog for nondeterministic programs.

1 Introduction

There is high demand for advanced information services in various application domains such as medical services and supply-chain management, as information and communication technology penetrates deeply into our society. Clinical decision support [1, 2] to prevent medical errors and order placement support for optimal inventory management [3] are typical examples. It is, however, not prudent to implement such functions as a normal part of the traditional information system using conventional programming languages. This is because expert knowledge is often large scale and complicated, and each application domain typically has its own specific structures and semantics. Therefore, not only the analysis, but also the description, audit, and maintenance of such knowledge are often difficult without expertise in the application domain. It is thus, essential to realize such advanced functions to allow domain experts themselves to describe, audit, and maintain their knowledge. A knowledge-based system approach is suitable for such purposes because a suitable framework for representing and managing expert knowledge is supplied.

Previously, Nagasawa et al. proposed the knowledge representation language DSP [4, 5] and its supporting environment. DSP is a nondeterministic functional language based on attribute grammars [6, 7] and is suitable for representing complex search problems without relying on any side effects. The supporting environment has been developed on top of an integrated development environment called Inside Prolog [8]. Inside Prolog provides standard Prolog functionality, conforming to ISO/IEC 13211-1 [9], and also a large variety of Application Programming Interfaces (APIs) that are essential for practical application development and multi-thread capability for enterprise use [10].

These features allow the consistent development of knowledge-based systems from prototypes to practical systems for both stand-alone and enterprise use [11]. Such systems have been applied to several practical applications, and the effectiveness thereof has been clarified. However, several issues have also been perceived from these experiences. One is the complexity of combining a Prolog-based system with a system written in a normal procedural language, such as Java. The other is the adaptability to a new computer environment such as mobile devices.

This paper describes the implementation techniques required to translate a nondeterministic functional language based on attribute grammars into a procedural language such as Java. The proposed techniques

are based on the techniques for Prolog to Java translation. Section 2 gives an overview of the knowledge representation language DSP, and clarifies how it differs from Prolog. In Section 3, the translation techniques for logic programming languages are briefly reviewed, and basic ideas useful for the translation of DSP identified. Section 4 discusses the program representations of DSP in Java, while Section 5 evaluates the performance using an experimental translator.

2 Overview of Knowledge Representation Language DSP

2.1 Background

It is essential to formally analyze, systematize, and describe the knowledge of an application domain in the development of a knowledge-based system. The description of knowledge is conceptually possible in any conventional programming language. Nevertheless, it is difficult to describe, audit, and maintain a knowledge base using a procedural language such as Java. This is because the knowledge of an application domain is often large scale and complicated, and each application domain has its own specific structures and semantics. In particular, the audit and maintenance of written knowledge is a major issue in an information system involving expert knowledge, because such a system is very often stiffened and the transfer of expert knowledge to succeeding generations is difficult [12]. Therefore, it is very important to provide a framework to enable domain experts themselves to describe, audit, and maintain their knowledge included in an information system [13]. It is perceived that a description language that is specific to an application domain and is designed so as to be described by domain experts is superior in terms of the minimality, constructibility, comprehensibility, extensibility, and formality of the language [14]. For this reason, Prolog cannot be considered as a candidate for a knowledge representation language.

DSP is a knowledge representation language based on nondeterministic attribute grammars. It is a functional language with a search capability using the generate and test method. Because the language is capable of representing trial and error without any side-effects or loop constructs, and the knowledge descriptions can be declaratively read and understood, it is suitable for representing domain-specific expert knowledge involving search problems.

2.2 Syntax and Semantics of DSP

A program unit to represent knowledge in DSP is called a “module”, and it represents a nondeterministic function involving no side-effects. Inherited attributes, synthesized attributes, and tentative variables for the convenience of program description, all of which are called variables, follow the single assignment rule and the assignment is unidirectional. Therefore, the computation process of a module can be represented as non-cyclic dependencies between variables.

Table 1. Typical statements in the DSP language

Type	Statement	Function
generator	`for(B,E,S)`	Assume a numeric value from B to E with step S
generator	`select(L)`	Assume one of the elements of a list L
generator	`call(M,I,O)`	Call a module M nondeterministically with inputs I and outputs O
calculator	`dcall(M,I,O)`	Call a module M deterministically with inputs I and outputs O
calculator	`find(M,I,OL)`	Get a list OL of all outputs of a module M with inputs I
tester	`when(C)`	Specify the domain C of a method
tester	`test(C)`	Specify the constraint C of a method
tester	`verify(C)`	Specify the verification condition C

Table 1 shows some typical statements in the language. In this table, the types, generator, calculator, and tester, are functional classifications in the generate and test method. Generators `for(B,E,S)`

and `select(L)` are provided as primitives for the convenience of knowledge representation although they can be defined as modules using the nondeterministic features of the language. Both `call(M,I,O)` and `dcall(M,I,O)` are used for module decomposition, with the latter restricting the first solution of a module call like `once/1` in Prolog [3], while the former calls a module nondeterministically. Calculator `find(M,I,OL)` collects all outputs of a module and returns a list thereof. Testers `when(C)` and `test(C)` are used to represent decomposition conditions. Both behaves in the same way in normal execution mode [4], although the former is intended to describe a guard of a method, while the latter describes a constraint. Tester `verify(C)` does not affect the execution of a module although it is classified as the tester. Solutions in which a verification condition is not satisfied are indicated as such, and these verification statuses are used to evaluate the inference results.

```
pointInQuarterCircle({R : real},           --(a)
                     {X : real, Y : real}) --(b)
  method
    X : real = for(0.0, R, 1.0);           --(c)
    Y : real = for(0.0, R, 1.0);           --(d)
    D : real = sqrt(X^2 + Y^2);            --(e)
    test(D =< R);                          --(f)
  end method;
end module;
```

Fig. 1. Module `pointInQuarterCircle`, which enumerates all points in a quarter circle

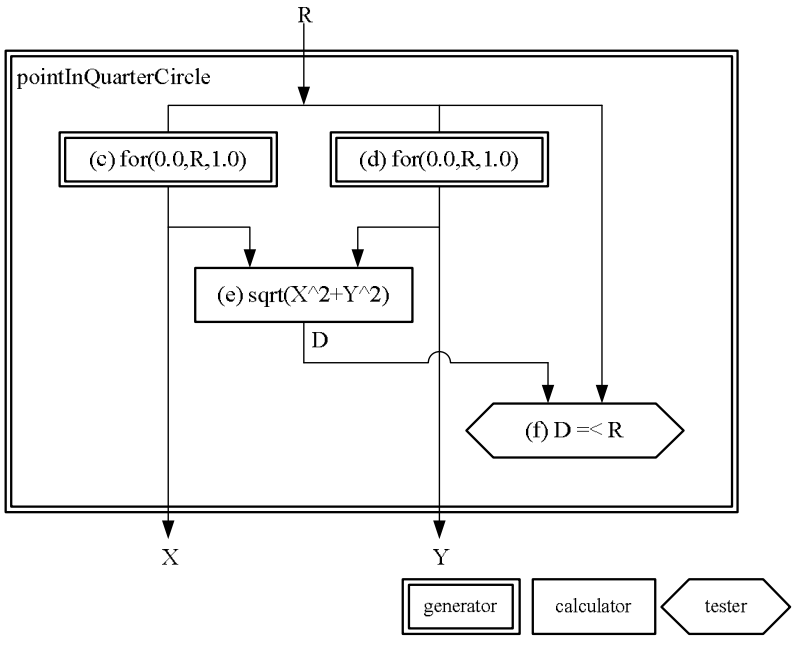


Fig. 2. Data flow diagram of module `pointInQuarterCircle`

[3] `dcall` stands for deterministic call.

[4] Failures of `when(C)` and `test(C)` are treated differently in debugging mode because of their semantic differences.

Figure 1 gives the code for module `pointInQuarterCircle`, which enumerates all points in a quarter circle with radius `R`. Statements (a) and (b) in Fig. 1 define the input and output variables of module `pointInQuarterCircle`, respectively. Statements (c) and (d) assume the values of the variables `X` and `Y` from `1` to `R` with an incremental step `1`. Statement (e) calculates the distance `D` between point `(0,0)` and point `(X,Y)`. Statement (f) checks if point `(X,Y)` is within the circle of radius `R`. Module `pointInQuarterCircle` runs nondeterministically for a given `R`, and returns one of all possible `{X,Y}` values [5]. Therefore, this module also behaves as a generator. Statements (c) to (f) can be listed in any order, and they are executed according to the dependencies between variables. Therefore, the computation process can be described as a non-cyclic data flow. Figure 2 shows the data flow diagram for module `pointInQuarterCircle`. Because no module includes any side-effects, the set of points returned by the module for the same input is always the same.

Figure 3 shows an example of module `for`, which implements the generator primitive `for`. If multiple methods are defined in a module with some overlap in their domains specified by `when`, the module works nondeterministically, and thus a module can also be a generator. In this example, there is overlap between the domains specified by statements `(a)` and `(c)`.

```
for({B : real, E : real, S : real},{N : real})
  method                          --The fist method
    when(B =< E);                 --(a)
    N : real = B;                 --(b)
  end method;
  method                          --The second method
    when(B+S =< E);               --(c)
    B1 : real = B+S;              --(d)
    call(for, {B1, E, S}, {N}); --(e)
  end method;
end;
```

Fig. 3. Module `for`, which implements the generator primitive `for`

2.3 Execution Model for DSP

Since the variables follow the single assignment rule and the assignment is unidirectional, the statements are partially ordered according to the dependencies between variables. In the execution, the statements must be totally reordered and evaluated in this order. Although the method used to order the partially ordered statements totally does not affect the set of solutions, the order of the generators affects the order of the solutions returned from a nondeterministic module.

The execution model for DSP can be represented in Prolog. Figure 4 illustrates an example of a simplified DSP interpreter in Prolog. In this interpreter, statements are represented as terms concatenated by ";" and it is assumed that the statements are totally ordered. Variables are represented using logical variables in Prolog. Actually, the development environment for DSP provides a compiler that translates into Prolog code, with the generated Prolog code translated into bytecode by the Prolog compiler in the runtime environment.

3 Translation Techniques for Logic Programming Languages

Prolog is a logic programming language that offers both declarative features and practical applicability to various application domains. Many implementation techniques for Prolog and its family have been proposed, while abstract machines represented by the WAM (Warren's Abstract Machine) [15] have proven

[5] `{X,Y}` represents a vector of two elements `X` and `Y`.

```
solve((A ; B)) :-
    solve(A),
    solve(B).
solve(call(M, In, Out)) :-
    reduce(call(M, In, Out), Body),
    solve(Body).
solve(dcall(M, In, Out)) :-
    reduce(call(M, In, Out), Body),
    solve(Body),!.
solve(find(M, In, OutList)) :-
    findall(Out, solve(M, In, Out), OutList).
solve(when(Exp)) :-
    call(Exp),!.
solve(test(Exp)) :-
    call(Exp),!.
solve(V := for(B, E, S)) :- !,
    for(B, E, S, V).
solve(V := select(L)) :- !,
    member(V, L).
solve(V := Exp) :-
    V is Exp.
```

Fig. 4. Simplified DSP interpreter in Prolog

effective practical implementation techniques. On the other hand, few Prolog implementations provide practical functionality applicable to both stand-alone systems and enterprise-mission-critical information systems without using other languages. Practically, Prolog is often combined with a conventional procedural language, such as Java, C, and C#, for use in practical applications. In such cases, language interoperability is an important issue.

Language translation is one possible solution for improving the interoperability between Prolog and other combined languages. jProlog [16] and Prolog Cafe [17] are Prolog to Java translators based on binarization [18], while P# [19] is a Prolog to C# translator based on Prolog Cafe with concurrent extensions. The binarization with continuation passing is a useful idea for handling nondeterminism simply in procedural languages. For example, the following clauses

```
p(X) :- q(X, Y), r(Y).
q(X, X).
r(X).
```

can be represented by semantically equivalent clauses that take a continuation goal `Cont` as the last parameter:

```
p(X, Cont) :- q(X, Y, r(Y, Cont)).
q(X, X, Cont) :- call(Cont).
r(X, Cont) :- call(Cont).
```

Once clauses have been transformed into this form, clauses composing a predicate can be translated into Java classes. Figure 5 gives an example of code generated by Prolog Cafe. Predicate `p/2` after binarization is represented as a Java class called `PRED_p_1`, which is a subclass of class `Predicate`. The parameters of a predicate call are passed as the arguments of the constructor of a class, while the right hand side of a clause is expanded as method `exec`.

If a predicate consists of multiple clauses as in the following predicate `p/1`, it may have choice points.

```
p(X) :- q(X, Y), r(Y).
p(X) :- r(X).
```

```
public class PRED_p_1 extends Predicate {
    public Term arg1;

    public PRED_p_1(Term a1, Predicate cont) {
        arg1 = a1;
        this.cont = cont; /* this.cont is inherited. */
    }
    ...
    public Predicate exec(Prolog engine) {
        engine.setB0();
        Term a1, a2;
        Predicate p1;
        a1 = arg1;
        a2 = new VariableTerm(engine);
        p1 = new PRED_r_1(a2, cont);
        return new PRED_q_2(a1, a2, p1);
    }
}
```

Fig. 5. Java code generated by Prolog Cafe

In such a case, the generated code becomes more complex than before because the choice points of `p/1` must be handled for backtracking. Figure 6 gives an example of the generated code for predicate `p/1` in the previous example. Each clause of a predicate is mapped to a subclass of a class representing the predicate. In this example, classes `PRED_p_1_1` and `PRED_p_1_2` correspond to the two clauses of predicate `p/1`. Methods `jtry` and `trust` of the Prolog engine correspond to WAM instructions that manipulate stacks and choice points for backtracking. The key ideas in Prolog Cafe are that continuation is represented as an instance of a Java class representing a predicate, and the execution control including backtracking follows the WAM. The translation is straightforward through the WAM, while the interoperability with Java-based systems is somewhat improved. On the other hand, the disadvantage is the performance of the generated code.

4 Program Representation in Java and Inference Engine

This section describes the translation techniques for the nondeterministic functional language DSP into Java based on the translation techniques for Prolog. Current implementations of the compiler and inference engine for DSP have been developed on top of Inside Prolog with the compiler generating Prolog code. Therefore, it is possible to translate this generated Prolog code into Java using Prolog Cafe. However, there are several differences between DSP and Prolog in terms of the semantics of variables and the determinism of statements. These differences allow several optimizations in performance, and the generated code can run faster than the code generated by Prolog Cafe for compatible Prolog programs. Fundamental ideas of our translation techniques utilize the single and unidirectional assignment features of variables and the deterministic features of some statements.

The overall structure of the Java code translated from DSP provides for one module being mapped to a single Java class, and each method in a module mapped to a single inner class of the class. Figure 7 shows an example of Java code for module `pointInQuarterCircle` given in Fig. 1. Inner classes are used to represent an execution context of a predicate as an internal state of a class instance. Therefore, the instances of an inner class are not declared as static unlike classes in Fig. 6.

An overview of the translation process follows. First, the data flow of a module is analyzed for each method based on the dependencies between variables, and the statements are reordered according to the analysis results. Next, the statements are grouped into translation units called continuation units, and Java code is generated for each method according to the continuation units.

```
public class PRED_p_1 extends Predicate {
    static Predicate _p_1_sub_1 = new PRED_p_1_sub_1();
    static Predicate _p_1_1 = new PRED_p_1_1();
    static Predicate _p_1_2 = new PRED_p_1_2();
    public Term arg1;

    ...
    public Predicate exec(Prolog engine) {
        engine.aregs[1] = arg1;
        engine.cont = cont;
        engine.setB0();
        return engine.jtry(_p_1_1, _p_1_sub_1);
    }
}

class PRED_p_1_sub_1 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        return engine.trust(_p_1_2);
    }
}

class PRED_p_1_1 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        Term a1, a2;
        Predicate p1;
        Predicate cont;
        a1 = engine.aregs[1];
        cont = engine.cont;
        a2 = new VariableTerm(engine);
        p1 = new PRED_r_1(a2, cont);
        return new PRED_q_2(a1, a2, p1);
    }
}

class PRED_p_1_2 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        Term a1;
        Predicate cont;
        a1 = engine.aregs[1];
        cont = engine.cont;
        return new PRED_r_1(a1, cont);
    }
}
```

Fig. 6. Java code with choice points generated by Prolog Cafe

4.1 Data Flow Analysis

As described in Sect. 2, it is necessary to reorder and evaluate statements so as to fulfill variable dependencies since statements can be listed in any order. Therefore, partially ordered statements must first be totally reordered. In the reordering process, the order of the generators should be kept as long as the variable dependencies are satisfied, because the order of generators affects the order of the solutions as described in Sec. 2. On the other hand, calculators or testers can be moved forward for the least commitment as long as partial orders are kept.

4.2 Continuation Unit

If statements of a method are totally ordered, they can be divided into several groups of statements. Each group is called a continuation unit and consists of a series of deterministic statements, such as calculators and testers, followed by a single generator. It should be noted that a continuation unit may not contain a generator if it is the last one in a method. In the translation, a continuation unit is treated as a unit to translate, and is mapped to a Java class representing a continuation.

In the example in Fig. 7, module `pointInQuarterCircle` has one method, and there are three continuation units in the method. Inner class `Method_1` corresponds to this method of the module, and class `Method_1_cu1` corresponds to the continuation unit for statement (c), class `Method_1_cu2` to one for statement (d), and class `Method_1_cu3` to one for statements (e) and (f), respectively.

4.3 Variable and Parameter Passing

Although variables follow the single assignment rule like Prolog, the binding of a variable is unidirectional unlike Prolog. Therefore, it is not necessary to introduce logical variables and unification, unlike in Prolog Cafe. This also implies that the trail stack and variable unbinding using the stack are unnecessary on backtracking. Therefore, a class representing the variables is only necessary as a place holder for the output values of a module. Class `Variable` is introduced to represent such variables.

Prolog Cafe uses the registers of the Prolog VM to manage the arguments of a goal. This approach is consistent with the WAM, but is sometimes inefficient since it requires arguments to be copied from/to registers to/from the stack on calls and backtracking. On the other hand, because the direction of variable binding is clearly defined in DSP, it is unnecessary to restore variable bindings on backtracking as described before. Instead, variables can always be overwritten when a goal is re-executed after backtracking. Therefore, input and output parameters can be passed as arguments of a class constructor. This simplifies the management of variables and arguments. In addition, as shown in Fig. 7, basic Java types, such as `int` and `double`, can be passed directly as inputs in some cases. This contributes to the performance improvement.

4.4 Inference Engine

An inference engine for the translated code is very simple because management of variables and registers on backtracking is unnecessary. Figure 8 shows an example of the inference engine called `VM`, which uses a stack represented as an array of interface `Executable` to store choice points. Method `call()` is an entry point to call the module to find an initial solution, while method `redo()` is used to find the next solution. A typical call procedure of a client program in Java is given below.

```
VM vm = new VM();
Double r = new Double(10.0);
Variable x = new Variable();
Variable y = new Variable();
Executable m = new PointInQuarterCircle(r, x, y,
                                        Executable.success);
for (boolean s = vm.call(m); s == true; s = vm.redo()) {
  System.out.println("X=" + x.doubleValue() +
                     ", Y=" + y.doubleValue());
}
```

```
public class PointInQuarterCircle implements Executable {
  private Double r;
  private Variable x;
  private Variable y;
  private Executable cont;
  public PointInQuarterCircle(Double r,
                              Variable x, Variable y, Executable cont)
  {
    this.r = r;
    this.x = x;
    this.y = y;
    this.cont = cont;
  }

  public Executable exec(VM vm) {
    return (new Method_1()).exec(vm);
  }

  public class Method_1 implements Executable {
    private Variable d = new Variable();
    private Executable method_1_cu1 = new Method_1_cu1();
    private Executable method_1_cu2 = new Method_1_cu2();
    private Executable method_1_cu3 = new Method_1_cu3();

    public Executable exec(VM vm) {
      return method_1_cu1.exec(vm);
    }

    class Method_1_cu1 implements Executable {
      public Executable exec(VM vm) {
        return new ForDouble(0.0, r.doubleValue(), 1.0, x, method_1_cu2);
      }
    }

    class Method_1_cu2 implements Executable {
      public Executable exec(VM vm) {
        return new ForDouble(0.0, r.doubleValue(), 1.0, y, method_1_cu3);
      }
    }

    class Method_1_cu3 implements Executable {
      public Executable exec(VM vm) {
        d.setValue(Math.sqrt(x.doubleValue()*x.doubleValue() +
                             y.doubleValue()*y.doubleValue()));
        if(!(d.doubleValue() <= r.doubleValue())){
          return Executable.failure;
        }
        return cont;
      }
    }
  }
}
```

Fig. 7. Java code generated for module `pointInQuarterCircle`

This client program creates an inference engine, prepares output variables to receive the values of a solution, creates an instance of class `PointInQuarterCircle` with inputs and outputs, and calls `call()` to find an initial solution. It then calls `redo()` to find the next one until there are no more solutions.

Because the implementation of the inference engine is simple and multi-thread safe, and the generated classes of a module are also multi-thread safe, it is easy to deploy instances of the engine in a multi-thread environment.

```
public class VM {
  private Executable[] choicepoint;
  private int ccp = -1; // Current choice point.
  ...

  public VM(int initSize) {
    choicepoint = new Executable[initSize];
  }
  ...
  public boolean call(Executable goal) {
    while (goal != null) {
      goal = goal.exec(this);
      if (goal == Executable.success) {
        return true;
      } else if (goal == Executable.failure) {
        goal = getChoicePoint();
      }
    }
    return false;
  }

  public boolean redo() {
    return call(getChoicePoint());
  }
}
```

Fig. 8. Inference engine for DSP

5 Implementation and Performance Evaluation

We have implemented the translator for DSP into Java based on the techniques proposed in Sec. 4. The translator is written in DSP itself and generates Java code.

Table 2 shows the performance results of 6 sample programs executed under Windows Vista on an Intel Core2Duo 2.53 GHz processor with 3.0 GB memory. Java 1.6, Prolog Cafe 1.2.5, and B-Prolog 7.4 [20] were used in the experiments. Because the Java garbage collector affects the performance, 512 MB memory was statically allocated for the heap in all cases except for one [6].

Program `plan` is a simple architecture design program for a parking structure. It can enumerate all possible column layouts for the given design conditions, such as free land space and the number of stories. Programs `nqueens`, `ack`, and `tarai` are well-known benchmarks, with `ack` and `tarai` using green cuts for guards in Prolog, while `ack w/o cuts` and `tarai w/o cuts` do not use cuts for guards. In the case of DSP, `ack` and `tarai` use `dcall` for self-recursive calls not to leave choice points, while `ack w/o cuts` and `tarai w/o cuts` use `call`. The programs written in DSP are compiled into Prolog and then compiled into bytecode. The programs are forced to backtrack in each iteration to enumerate all solutions, and the execution times in milliseconds are averages over 10 trials.

[6] About 1000 MB was allocated for the generated code for `tarai w/o cuts`.

These results show that the proposed translator generates over 25 times faster code than Prolog Cafe, over 2 times faster code than B-Prolog, and over 5 times faster code than DSP on top of Inside Prolog for `plan` and `nqueens`. On the other hand, for `ack` and `tarai` the translator generates about 2 to 3 times faster code than Prolog Cafe, but about 5 to 15 times slower code than B-Prolog. The translator also generates about 8 to 13 times faster code than Prolog Cafe, but about 4 to 10 times slower code than B-Prolog for `ack w/o cuts` and `tarai w/o cuts`. Here, `plan` and `nqueens` are nondeterministic, while `ack` and `tarai` are deterministic. `ack w/o cuts` and `tarai w/o cuts` are also deterministic, but they involve backtracking because of the lack of green cuts.

These experiments indicate that the proposed translation techniques can generate faster code than Prolog Cafe and DSP on top of Inside Prolog for all 6 programs, and faster code than B-Prolog for nondeterministic programs. In the case of deterministic programs, the advantage of the proposed translation techniques is obvious against Prolog Cafe if green cuts are not used in Prolog. The reason why these distinctive differences are observed seems to be that the simplification of the variable and register management for backtracking contributes to the performance improvement of nondeterministic programs, but it is not effective for deterministic programs with green cuts.

In the case of B-Prolog, the execution time of `tarai` is almost the same as that of `tarai w/o cuts`. This is because B-Prolog compiler reduces choice points using matching trees for both `tarai` and `tarai w/o cuts` [21]. Although the DSP language has no explicit cut operator of Prolog, improving the performance by inserting cut instructions automatically in the case of exclusive `when` conditions is a future issue.

The number of instances created during an execution has a negative impact on performance because of the garbage collection. Obviously, the number of instances created by the generated code for the proposed translation techniques is greater than that for Prolog Cafe. In the case of `tarai w/o cuts`, the generated code requires more memory than others to prevent the garbage collection. In the example in Fig. 7, it is clear that the number of instances can be reduced by merging class `Method_1_cu1` with class `Method_1`. Improving the performance by the reduction of instance creation is an important future issue.

Table 2. Experimental results (in milliseconds)

Program	DSP on Prolog	B-Prolog	Prolog Cafe	Translator
plan	685.0	295.1	2519.4	90.5
nqueens	594.9	296.2	3279.2	120.3
ack	1568.2	52.9	990.7	265.0
tarai	1302.7	49.4	1680.1	740.8
ack w/o cuts	2035.1	104.7	3421.3	403.9
tarai w/o cuts	1307.8	49.2	6282.2	489.5

6 Conclusions

This paper described the techniques for translating the nondeterministic functional language DSP based on attribute grammars into Java. The DSP is designed for knowledge representation of large scale and complicated expert knowledge in application domains. It is capable of representing trial and error without any side-effects or loop constructs using nondeterministic features. Current development and runtime environments are built on top of Inside Prolog, while the runtime environment can be embedded in a Java-based application server. However, issues regarding language interoperability and adaptability to new computer environments are envisaged when applied to practical application development. The language translation is intended to improve the interoperability and adaptability of DSP.

The proposed translation techniques are based on binarization and the techniques proposed for the translation of Prolog. The performance, however, is improved by introducing the continuation unit and simplifying the management of variables and registers using the semantic differences of variables and

explicit determinism of some statements. An experimental translator written in DSP itself generates Java code from DSP descriptions, and the experimental results indicate that the generated code is over 25 times faster than that of Prolog Cafe for nondeterministic programs, and over 2 times faster for deterministic programs. The generated code is also over 2 times faster than B-Prolog for nondeterministic programs. However, the generated code is about 3 to 15 times slower than B-Prolog for deterministic programs. Improving the performance of deterministic programs is an important future issue.

References

1. Kaplan, B.: Evaluating informatics applications – clinical decision support systems literature review. International Journal of Medical Informatics **64** (2001) 15–37
2. Takada, A., Nagase, K., Ohno, K., Umeda, M., Nagasawa, I.: Clinical decision support system, how do we realize it for hospital information system? Japan journal of medical informatics **27** (2007) 315–320
3. Nagasawa, H., Nagasawa, I., Takata, O., Umeda, M., Hashimoto, M., Takizawa, C.: Knowledge modeling for operation management support of a distribution center. In: The Sixth IEEE International Conference on Computer and Information Technology (CIT2006). (2006) 6
4. Nagasawa, I., Maeda, J., Tegoshi, Y., Makino, M.: A programming technique for some combination problems in a design support system using the method of generate-and-test. Journal of Structural and Construction Engineering (1990)
5. Umeda, M., Nagasawa, I., Higuchi, T.: The elements of programming style in design calculations. In: Proceedings of the Ninth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. (1996) 77–86
6. Katayama, T.: A computation model based on attribute grammar. Journal of Information Processing Society of Japan **24** (1983) 147–155
7. Deransart, P., Jourdan, M., eds.: Attribute Grammars and their Applications. Number 461 in Lecture Notes in Computer Science. Springer-Verlag (1990)
8. Katamine, K., Umeda, M., Nagasawa, I., Hashimoto, M.: Integrated development environment for knowledge-based systems and its practical application. IEICE Transactions on Information and Systems **E87-D** (2004) 877–885
9. ISO/IEC: 13211-1 Information technology – Programming Languages – Prolog – Part 1: General core. (1995)
10. Umeda, M., Katamine, K., Nagasawa, I., Hashimoto, M., Takata, O.: Multi-threading inside prolog for knowledge-based enterprise applications. In: Declarative Programming for Knowledge Management. Volume 4369 of Lecture Notes in Computer Science., Springer Science+Business Media (2006) 200–214
11. Umeda, M., Katamine, K., Nagasawa, I., Hashimoto, M., Takata, O.: The design and implementation of knowledge processing server for enterprise information systems. Transactions of Information Processing Society of Japan **48** (2007) 1965–1979
12. Nagasawa, I.: Feature of design and intelligent CAD. Journal of The Japan Society for Precision Engineering **54** (1988) 1429–1434
13. Umeda, M., Mure, Y.: Knowledge management strategy and tactics for forging die design support. In: The Proceedings of the 18th International Conference on Applications of Declarative Programming and Knowledge Management. (2009) 21–36
14. Hirota, T., Hashimoto, M., Nagasawa, I.: A discussion on conceptual model description language specific for an application domain. Transactions of Information Processing Society of Japan **36** (1995) 1151–1162
15. Ait-Kaci, H.: Warren's Abstract Machine. The MIT Press (1991)
16. Demoen, B., Tarau, P.: jprolog home page. http://www.cs.kuleuven.ac.be/ bmd/PrologInJava/(1996)
17. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe: A prolog to java translator system. In: The Proceedings of the 16th International Conference on Applications of Declarative Programming and Knowledge Management. (2005) 45–54
18. Tarau, P., Boyer, M.: Elementary logic programs. In: Programming Language Implementation and Logic Programming. Volume 456 of Lecture Notes in Computer Science. (1990) 159–173
19. Cook, J.J.: Language Interoperability and Logic Programming Languages. Doctor of philosophy, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh (2004)
20. Neng-Fa, Z.: The language features and architecture of b-prolog. Theory and Practice of Logic Programming (2011)
21. Neng-Fa, Z.: Global optimizations in a prolog compiler for the TOAM. J. Logic Programming (1993) 265–294

Sensitivity Analysis for Declarative Relational Query Languages with Ordinal Ranks*

Radim Belohlavek, Lucie Urbanova, and Vilem Vychodil

DAMOL (Data Analysis and Modeling Laboratory)
Dept. Computer Science, Palacky University, Olomouc
17. listopadu 12, CZ–77146 Olomouc, Czech Republic
radim.belohlavek@acm.org, lucie.urbanova01@upol.cz, vychodil@acm.org

Abstract. We present sensitivity analysis for results of query executions in a relational model of data extended by ordinal ranks. The underlying model of data results from the ordinary Codd's model of data in which we consider ordinal ranks of tuples in data tables expressing degrees to which tuples match queries. In this setting, we show that ranks assigned to tuples are insensitive to small changes, i.e., small changes in the input data do not yield large changes in the results of queries.

Keywords: declarative query languages, ordinal ranks, relational databases, residuated lattices

1 Introduction

Since its inception, the relational model of data introduced by E. Codd [10] has been extensively studied by both computer scientists and database systems developers. The model has become the standard theoretical model of relational data and the formal foundation for relational database management systems. Various reasons for the success and strong position of Codd's model are analyzed in [14], where the author emphasizes that the main virtues of the model like logical and physical data independence, declarative style of data retrieval (database querying), access flexibility and data integrity are consequences of a close connection between the model and the first-order predicate logic.

This paper is a continuation of our previous work [4, 5] where we have introduced an extension of Codd's model in which tuples are assigned ordinal ranks. The motivation for the model is that in many situations, it is natural to consider not only the exact matches of queries in which a tuple of values either *does* or *does not* match a query Q but also approximate matches where tuples match queries to degrees. The degrees of approximate matches can usually be described verbally using linguistic modifiers like "not at all (matches)" "almost (matches)", "more or less (matches)", "fully (matches)", etc. From the user's point of view, each data table in our extended relational model consists of (i) an ordinary data table whose meaning is the same as in the Codd's model and (ii) ranks assigned to all tuples in the original data table. This way, we come up with a notion of a ranked data table (shortly, an RDT). The ranks in RDTs are interpreted as "goodness of match" and the interpretation of RDTs is the same as in the Codd's model—they represent answers to queries which are, in addition, equipped with priorities expressed by the ranks. A user who looks at an answer to a query in our model is typically looking for the best match possible represented by a tuple or tuples in the resulting RDT with the highest ranks (i.e., highest priorities).

In order to have a suitable formalization of ranks and to perform operations with ranked data tables, we have to choose a suitable structure for ranks. Since ranks are meant to be compared by users, the set L of all considered ranks should be equipped with a partial order $\leq$, i.e. $\langle L, \leq \rangle$ should be a poset. Moreover, it is convenient to postulate that $\langle L, \leq \rangle$ is a complete lattice [7], i.e., for each subset $A \subseteq L$, its least upper bound (a supremum) and greatest lower bound (an infimum) exist. This way, for any $A \subseteq L$, one can take the least rank in L which represents a higher priority (a better match) than all ranks from A. Such a rank is then the supremum of A (dually for the infimum). Since $\langle L, \leq \rangle$ is a complete lattice, it contains the least element denoted 0 (no match at all) and the greatest element denoted 1 (full match).

The set L of all ranks should also be equipped with additional operations for aggregation of ranks. Indeed, if tuple t with rank a is obtained as one of the results of subquery Q_1 and the same t with another

* Supported by grant no. P103/11/1456 of the Czech Science Foundation.

rank b is obtained from answers to subquery Q_2 then we might want to express the rank to which t matches a compound conjunctive query "Q_1 *and* Q_2". A natural way to do so is to take a suitable binary operation $\otimes\colon L \times L \to L$ which acts as a conjunctor and take $a \otimes b$ for the resulting rank. Obviously, not every binary operation on L represents a (reasonable) conjunctor, i.e. we may restrict the choices only to particular binary operations that make "good conjunctors". There are various ways to impose such restrictions. In our model, we follow the approach of using residuated conjunctions that has proved to be useful in logics based on residuated lattices [2, 18, 19]. Namely, we assume that $\langle L, \otimes, 1\rangle$ is a commutative monoid (i.e., $\otimes$ is associative, commutative, and neutral with respect to 1) and there is a binary operation $\to$ on L such that for all $a, b, c \in L$:

$$a \otimes b \leq c \quad \text{if and only if} \quad a \leq b \to c. \tag{1}$$

Operations $\otimes$ (a multiplication) and $\to$ (a residuum) satisfying (1) are called *adjoint operations.* Altogether, the structure for ranks we use is a *complete residuated lattice* $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \to, 0, 1\rangle$, i.e., a complete lattice in which $\otimes$ and $\to$ are adjoint operations, and $\wedge$ and $\vee$ denote the operations of infimum and supremum, respectively. Considering $\mathbf{L}$ as a basic structure of ranks brings several benefits. First, in multiple-valued logics and in particular fuzzy logics [18, 19], residuated lattices are interpreted as structures of truth degrees and the relationship (1) between $\otimes$ (a fuzzy conjunction) and $\to$ (a fuzzy implication) is derived from requirements on graded counterpart of the *modus ponens* deduction rule (currently, there are many strong-complete logics based on residuated lattices).

Remark 1. The graded counterpart of *modus ponens* [19, 26] can be seen as a generalized deduction rule saying "from φ valid (at least) to degree $a \in L$ and $\varphi \Rightarrow \psi$ valid (at least) to degre $b \in L$, infer ψ valid (at least) to degree $a \otimes b$". If if-part of (1) ensures that the rule is sound while the only-if part ensures that it is as powerful as possible, i.e., $a \otimes b$ is the highest degree to which we infer ψ valid provided that φ valid at least to degree a and $\varphi \Rightarrow \psi$ valid at least to degre $b \in L$. This relationship between $\to$ (a truth function for logical connective imlication $\Rightarrow$) and $\otimes$ has been discovered in [17] and later used, e.g., in [16, 26]. Interestingly, (1) together with the lattice ordering ensure enough properties of $\to$ and $\otimes$. For instance, $\to$ is antitone in the first argument and is monotone in the second one, condition $a \leq b$ iff $a \to b = 1$ holds for all $a, b \in L$, $a \to (b \to c)$ equals $(a \otimes b) \to c$ for all $a, b, c \in L$, etc. Since complete residuated lattices are in general weaker structures than Boolean algebras, not all laws satisfied by truth functions of the classic conjunction and implication are preserved by all complete residuated lattices. For instance, neither $a \otimes a = a$ (idempotency of $\otimes$) nor $(a \to 0) \to 0 = a$ (the law of double negation) nor $a \vee (a \to 0) = 1$ (the law of the excluded middle) hold in general. Nevertheless, complete residuated lattices are strong enough to provide a formal framework for relational analysis and similarity-based reasoning as it has been shown by previous results.

Second, our extension of the Codd's model results from the model by replacing the two-element Boolean algebra, which is the classic structure of truth values, by a more general structure of truth values represented by a residuated lattice, i.e. we make the following shift in (the semantics of) the underlying logic:

$$\textit{two-element Boolean algebra} \quad \Longmapsto \quad \textit{a complete residuated lattice.}$$

Third, the original Codd's model is a special case of our model for $\mathbf{L}$ being the two-element Boolean algebra (only two borderline ranks 1 and 0 are available). As a practical consequence, data tables in the Codd's model can be seen as RDTs where all ranks are either equal to 1 (full match) or 0 (no match; tuples with 0 rank are considered as not present in the result of a query). Using residuated lattices as structures of truth degrees, we obtain a generalization of Codd's model which is based on solid logical foundations and has desirable properties. In addition, its relationship to residuated first-order logics is the same as the relationship of the original Codd's model to the classic first-order logic. The formalization we offer can further be used to provide insight into several isolated approaches that have been provided in the past, see e.g. [8], [15], [23], [27], [28], [30], and a comparison paper [6].

A typical choice of $\mathbf{L}$ is a structure with $L = [0, 1]$ (ranks are taken from the real unit interval), $\wedge$ and $\vee$ being minimum and maximum, $\otimes$ being a left-continuous (or a continuous) t-norm with the corresponding $\to$, see [2, 18, 19]. For example, an RDT with ranks coming from such $\mathbf{L}$ is in Table 1. It can be seen as a

Table 1. Houses for sale at $200,000 with square footage 1200

	agent	*id*	*sqft*	*age*	*location*	*price*
0.93	Brown	138	1185	48	Vestal	$228,500
0.89	Clark	140	1120	30	Endicott	$235,800
0.86	Brown	142	950	50	Binghamton	$189,000
0.85	Brown	156	1300	85	Binghamton	$248,600
0.81	Clark	158	1200	25	Vestal	$293,500
0.81	Davis	189	1250	25	Binghamton	$287,300
0.75	Davis	166	1040	50	Vestal	$286,200
0.37	Davis	112	1890	30	Endicott	$345,000

result of similarity-based query "show all houses which are sold for (approximately) $200,000 and have (approximately) 1200 square feet". The left-most column contains ranks. The remaining part of the table is a data table in the usual sense containing tuples of values. At this point, we do not explain in detail how the particular ranks in Table 1 have been obtained (this will be outlined in further sections). One way is by executing a similarity-based query that uses additional information about similarity (proximity) of domain values which is also described using degrees from $\mathbf{L}$. Note that the concept of a similarity-based query appears when human perception is involved in rating or comparing close values from domains where not only the exact equalities (matches) are interesting. For instance, a person searching in a database of houses is usually not interested in houses sold for a particular exact price. Instead, the person wishes to look at houses sold approximately at that price, including those which are sold for other prices that are sufficiently close. While the ranks constitute a "visible" part of any RDT, the similarities are not a direct part of RDT and have to be specified for each domain independently. They can be seen as an additional (background) information about domains which is supplied by users of the database system.

Let us stress the meaning of ranks as priorities. As it is usual in fuzzy logics in narrow sense, their meaning is primarily *comparative*, cf. [19, p. 2] and the comments on comparative meaning of truth degrees therein. In our example, it means that tuple $\langle$Clark, 140, 1120, 30, Endicott, $235,800$\rangle$ with rank 0.89 is a better match than tuple $\langle$Brown, 142, 950, 50, Binghamton, $189,000$\rangle$ whose rank 0.86 is strictly smaller. Thus, for end-users, the numerical values of ranks (if L is a unit interval) are not so important, the important thing is the relative ordering of tuples given by the ranks.

Note that our model which provides theoretical foundations for similarity-based databases [4, 5] should not be confused with models for probabilistic databases [29] which have recently been studied, e.g. in [9, 12, 13, 20, 22, 25], see also [11] for a survey. In particular, numerical ranks used in our model (if $L = [0, 1]$) cannot be interpreted as probabilities, confidence degrees of belief degrees as in case of probabilistic databases where ranks play such roles. In probabilistic databases, the tuples (i.e., the data itself) are uncertain and the ranks express probabilities that tuples appear in data tables. Consequently, a probabilistic database is formalized by a discrete probability space over the possible contents of the database [11]. Nevertheless, the underlying logic of the models is the classical two-valued first-order logic—only yes/no matches are allowed (with uncertain outcome). In our case, the situation is quite different. The data (represented by tuples) is absolutely certain but the tuples are allowed to match queries to degrees. This, translated in terms of logic, means that formulas (encoding queries) are allowed to be evaluated to truth degrees other than 0 and 1. Therefore, the underlying logic in our model is not the classic two-element Boolean logic as we have argued hereinbefore.

In [1], a report written by leading authorities in database systems, the authors say that the current database management systems have no facilities for either approximate data or imprecise queries. According to this report, the management of uncertainty and imprecision is one of the six currently most important research directions in database systems. Nowadays, probabilistic databases (dealing with approximate data) are extensively studied. On the contrary, it seems that similarity-based databases (dealing with imprecise queries) have not yet been paid full attention. This paper is a contribution to theoretical foundations of similarity-based databases.

2 Problem Setting

The issue we address in this paper is the following. In our model, we can get two or more RDTs (as results of queries) which are not exactly the same but which are perceived (by users) as being similar. For instance, one can obtain two RDTs containing the same tuples with numerical values of ranks that are almost the same. A question is whether such similar RDTs, when used in subsequent queries, yield similar results. In this paper, we present a preliminary study of the phenomenon of similarity of RDTs and its relationship to the similarity of query results obtained by applying queries to similar input data tables. We present basic notions and results providing formulas for computing estimations of similarity degrees. The observations we present provide a formal justification for the phenomenon discussed in the previous section—slight changes in ranks do not have a large impact on the results of (complex) queries. The results are obtained for any complete residuated lattice taken as the structure of ranks (truth degrees). Note that the basic query systems in our model are (extensions of) domain relational calculus [5, 24] and relational algebra [4, 24]. We formulate the results in terms of operations of the relational algebra but due to its equivalence with the domain relational calculus [5], the results pertain to both the query systems. Thus, based on the domain relational calculus, one may design a declarative query language preserving similarity in which execution of queries is based on transformations to expressions of relational algebra in a similar way as in the classic case [24].

The rest of the paper is organized as follows. Section 3 presents a short survey of notions. Section 4 contains results on sensitivity analysis, an illustrative example, and a short outline of future research. Because of the limited scope of the paper, proofs are sketched or omitted.

3 Preliminaries

In this section, we recall basic notions of RDTs and relational operations we need to provide insight into the sensitivity issues of RDTs in Section 4. Details can be found in [2, 4, 6]. In the rest of the paper, $\mathbf{L}$ always refers to a complete residuated lattice $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \rightarrow, 0, 1\rangle$, see Section 1.

3.1 Basic Structures

Given $\mathbf{L}$, we make use of the following notions: An $\mathbf{L}$-set A in universe U is a map $A\colon U \rightarrow L$, $A(u)$ being interpreted as "the degree to which u belongs to A". If $\mathbf{L}$ is the two-element Boolean algebra, then $A\colon U \rightarrow L$ is an indicator function of a classic subset of U, $A(u) = 1$ ($A(u) = 0$) meaning that u belongs (does not belong) to that subset. In our approach, we tacitly identify sets with their indicator functions. In a similar way, a binary $\mathbf{L}$-relation B on U is a map $B\colon U \times U \rightarrow L$, $B(u_1, u_2)$ interpreted as "the degree to which u_1 and u_2 are related according to B". Hence, B is an $\mathbf{L}$-set in universe $U \times U$.

3.2 Ranked Data Tables over Domains with Similarities

We denote by Y a set of *attributes*, any subset $R \subseteq Y$ is called a *relation scheme*. For each attribute $y \in Y$ we consider its *domain* D_y. In addition, each D_y is equipped with a binary $\mathbf{L}$-relation $\approx_y$ on D_y satisfying reflexivity ($u \approx_y u = 1$) and symmetry $u \approx_y v = v \approx_y u$ (for all $u, v \in D_y$). Each binary $\mathbf{L}$-relation $\approx_y$ on D_y satisfying (i) and (ii) shall be called a *similarity*. Pair $\langle D_y, \approx_y\rangle$ is called a *domain with similarity*.

Tuples contained in data tables will be considered as usual, i.e., as elements of Cartesian products of domains. Recall that a Cartesian product $\prod_{i\in I} D_i$ of an I-indexed system $\{D_i \mid i \in I\}$ of sets D_i ($i \in I$) is a set of all maps $t\colon I \rightarrow \bigcup_{i\in I} D_i$ such that $t(i) \in D_i$ holds for each $i \in I$. Under this notation, a *tuple* over $R \subseteq Y$ is any element from $\prod_{y\in R} D_y$. For brevity, $\prod_{y\in R} D_y$ is denoted by $\mathrm{Tupl}(R)$. Following the example in Table 1, tuple $\langle \texttt{Brown}, \texttt{142}, \texttt{950}, \texttt{50}, \texttt{Binghamton}, \texttt{\$189,000}\rangle$ is a map $r \in \mathrm{Tupl}(R)$ for $R = \{\texttt{agent}, \texttt{id}, \ldots, \texttt{price}\}$ such that $r(\texttt{agent}) = \texttt{Brown}$, $r(\texttt{id}) = \texttt{142}$, etc.

A *ranked data table* on $R \subseteq Y$ over $\{\langle D_y, \approx_y\rangle \mid y \in R\}$ (shortly, an RDT) is any (finite) $\mathbf{L}$-set $\mathcal{D}$ in $\mathrm{Tupl}(R)$. The degree $\mathcal{D}(r)$ to which r belongs to $\mathcal{D}$ is called a *rank* of tuple r in $\mathcal{D}$. According to its definition, if $\mathcal{D}$ is an RDT on R over $\{\langle D_y, \approx_y\rangle \mid y \in R\}$ then $\mathcal{D}$ is a map $\mathcal{D}\colon \mathrm{Tupl}(R) \rightarrow L$. Note that $\mathcal{D}$ is an n-ary $\mathbf{L}$-relation between domains D_y ($y \in Y$) since $\mathcal{D}$ is a map from $\prod_{y\in R} D_y$ to L. In our example, $\mathcal{D}(r) = 0.86$ for r being the tuple with $r(\texttt{id}) = \texttt{142}$.

3.3 Relational Operations with RDTs

Relational operations we consider in this paper are the following: For RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ on T, we put $(\mathcal{D}_1 \cup \mathcal{D}_2)(t) = \mathcal{D}_1(t) \vee \mathcal{D}_2(t)$ and $(\mathcal{D}_1 \cap \mathcal{D}_2)(t) = \mathcal{D}_1(t) \wedge \mathcal{D}_2(t)$ for each $t \in \mathrm{Tupl}(T)$; $\mathcal{D}_1 \cup \mathcal{D}_2$ and $\mathcal{D}_1 \cap \mathcal{D}_2$ are called the *union* and the $\wedge$-*intersection* of $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Analogously, one can define an $\otimes$-*intersection* $\mathcal{D}_1 \otimes \mathcal{D}_2$. Hence, $\cup$, $\cap$, and $\otimes$ are defined componentwise based on the operations of the complete residuated lattice $\mathbf{L}$.

Moreover, our model admits new operations that are trivial in the classic model. For instance, for $a \in L$, we introduce an *a-shift* $a{\to}\mathcal{D}$ of $\mathcal{D}$ by $(a{\to}\mathcal{D})(t) = a \to \mathcal{D}(t)$ for all $t \in \mathrm{Tupl}(T)$.

Remark 2. Note that if $\mathbf{L}$ is the two-element Boolean algebra then a-shift is a trivial operation since $1 \to \mathcal{D} = \mathcal{D}$ and $0 \to \mathcal{D}$ produces a possibly infinite table containing all tuples from $\mathrm{Tupl}(T)$. In our model, an a-shift has the following meaning: If $\mathcal{D}$ is a result of query Q then $(a{\to}\mathcal{D})(t)$ is a "degree to which t matches query Q at least to degree a". This follows from properties of residuum, see [2, 19]. Hence, a-shifts allow us to emphasize results that match queries at least to a prescribed degree a.

The remaining relational operations we consider represent counterparts of projection, selection, and join in our model. If $\mathcal{D}$ is an RDT on T, the *projection* $\pi_R(\mathcal{D})$ of $\mathcal{D}$ onto $R \subseteq T$ is defined by

$$(\pi_R(\mathcal{D}))(r) = \bigvee_{s \in \mathrm{Tupl}(T \setminus R)} \mathcal{D}(rs),$$

for each $r \in \mathrm{Tupl}(R)$. In our example, the result of $\pi_{\{\mathtt{location}\}}(\mathcal{D})$ is a ranked data table with single column such that $\pi_{\{\mathtt{location}\}}(\mathcal{D})(\langle \mathtt{Binghamton} \rangle) = 0.86$, $\pi_{\{\mathtt{location}\}}(\mathcal{D})(\langle \mathtt{Vestal} \rangle) = 0.93$, and $\pi_{\{\mathtt{location}\}}(\mathcal{D})(\langle \mathtt{Endicott} \rangle) = 0.89$.

A similarity-based selection is a counterpart to ordinary selection which selects from a data table all tuples which approximately match a given condition: Let $\mathcal{D}$ be an RDT on T and let $y \in T$ and $d \in D_y$. Then, a *similarity-based selection* $\sigma_{y \approx d}(\mathcal{D})$ of tuples in $\mathcal{D}$ matching $y \approx d$ is defined by

$$\big(\sigma_{y \approx d}(\mathcal{D})\big)(t) = \mathcal{D}(t) \otimes t(y) \approx_y d.$$

Considering $\mathcal{D}$ as a result of query Q, the rank of t in $\sigma_{y \approx d}(\mathcal{D})$ can be interpreted as a degree to which "t matches the query Q and the y-value of t is similar to d". In particular, an interesting case is $\sigma_{p \approx q}(\mathcal{D})$ where p and q are both attributes with a common domain with similarity.

Similarity-based joins are considered as derived operations based on Cartrsian products and similarity-based selections. For $r \in \mathrm{Tupl}(R)$ and $s \in \mathrm{Tupl}(S)$ such that $R \cap S = \emptyset$, we define a concatenation $rs \in \mathrm{Tupl}(R \cup S)$ of tuples r and s so that $(rs)(y) = r(y)$ for $y \in R$ and $(rs)(y) = s(y)$ for $y \in S$. For RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ on disjoint relation schemes S and T we define a RDT $\mathcal{D}_1 \times \mathcal{D}_2$ on $S \cup T$, called a *Cartesian product* of $\mathcal{D}_1$ and $\mathcal{D}_2$, by $(\mathcal{D}_1 \times \mathcal{D}_2)(st) = \mathcal{D}_1(s) \otimes \mathcal{D}_2(t)$. Using Cartesian products and similarity-based selections, we can introduce *similarity-based θ-joins* such as $\mathcal{D}_1 \bowtie_{p \approx q} \mathcal{D}_2 = \sigma_{p \approx q}(\mathcal{D}_1 \times \mathcal{D}_2)$. Various other types of similarity-based joins can be introduced in our model, see [5].

4 Estimations of Sensitivity of Query Results

4.1 Rank-Based Similarity of Query Results

We now introduce the notion of similarity of RDTs which is based on the idea that RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ (on the same relation scheme) are similar iff for each tuple t, ranks $\mathcal{D}_1(t)$ and $\mathcal{D}_2(t)$ are similar (degrees from $\mathbf{L}$). Similarity of ranks can be expressed by biresiduum $\leftrightarrow$ (a fuzzy equivalence [2, 18, 19]) which is a derived operation of $\mathbf{L}$ such that $a \leftrightarrow b = (a \to b) \wedge (b \to a)$. Since we are interested in similarity of $\mathcal{D}_1(t)$ and $\mathcal{D}_2(t)$ for all possible tuples t, it is straightforward to define the similarity $E(\mathcal{D}_1, \mathcal{D}_2)$ of $\mathcal{D}_1$ and $\mathcal{D}_2$ by an infimum which goes over all tuples:

$$E(\mathcal{D}_1, \mathcal{D}_2) = \bigwedge_{t \in \mathrm{Tupl}(T)} \big(\mathcal{D}_1(t) \leftrightarrow \mathcal{D}_2(t)\big). \tag{2}$$

An alternative (but equivalent) way is the following: we first formalize a degree $S(\mathcal{D}_1, \mathcal{D}_2)$ to which $\mathcal{D}_1$ is included in $\mathcal{D}_2$. We can say that $\mathcal{D}_1$ is fully included in $\mathcal{D}_2$ iff, for each tuple t, the rank $\mathcal{D}_2(t)$ is at least as high as the rank $\mathcal{D}_1(t)$. Notice that in the classic (two-values) case, this is exactly how one defines

the ordinary subsethood relation "$\subseteq$". Considering general degrees of inclusion (subsethood), a degree $S(\mathcal{D}_1, \mathcal{D}_2)$ to which $\mathcal{D}_1$ is included in $\mathcal{D}_2$ can be defined as follows:

$$S(\mathcal{D}_1, \mathcal{D}_2) = \bigwedge_{t \in \mathrm{Tupl}(T)} \big(\mathcal{D}_1(t) \rightarrow \mathcal{D}_2(t)\big). \tag{3}$$

It is easy to prove [2] that (2) and (3) satisfy:

$$E(\mathcal{D}_1, \mathcal{D}_2) = S(\mathcal{D}_1, \mathcal{D}_2) \wedge S(\mathcal{D}_2, \mathcal{D}_1). \tag{4}$$

Note that E and S defined by (2) and (3) are known as degrees of similarity and subsethood from general fuzzy relational systems [2] (in this case, the fuzzy relations are RDTs).

The following assertion shows that $\cup$, $\cap$, $\otimes$, and a-shifts preserve subsethood degrees given by (3). In words, the degree to which $\mathcal{D}_1 \cup \mathcal{D}_2$ is included in $\mathcal{D}'_1 \cup \mathcal{D}'_2$ is at least as high as the degree to which $\mathcal{D}_1$ is included in $\mathcal{D}'_1$ and $\mathcal{D}_2$ is included in $\mathcal{D}'_2$. A similar verbal description can be made for the other operations.

Theorem 1. *For any $\mathcal{D}_1$, $\mathcal{D}'_1$, $\mathcal{D}_2$, and $\mathcal{D}'_2$ on relation scheme T,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{D}'_1 \cup \mathcal{D}'_2), \tag{5}$$

$$S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cap \mathcal{D}_2, \mathcal{D}'_1 \cap \mathcal{D}'_2), \tag{6}$$

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \otimes \mathcal{D}_2, \mathcal{D}'_1 \otimes \mathcal{D}'_2), \tag{7}$$

$$S(\mathcal{D}_1, \mathcal{D}_2) \leq S(a \rightarrow \mathcal{D}_1, a \rightarrow \mathcal{D}_2). \tag{8}$$

Proof (sketch). (5): Using adjointness, it suffices to check that $\big(S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2)\big) \otimes (\mathcal{D}_1 \cup \mathcal{D}_2)(t) \leq (\mathcal{D}'_1 \cup \mathcal{D}'_2)(t)$ holds true for any $t \in \mathrm{Tupl}(T)$. Using (3), the monotony of $\otimes$ and $\wedge$ yields $\big(S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2)\big) \otimes (\mathcal{D}_1 \cup \mathcal{D}_2)(t) \leq \big((\mathcal{D}_1(t) \rightarrow \mathcal{D}'_1(t)) \wedge (\mathcal{D}_2(t) \rightarrow \mathcal{D}'_2(t))\big) \otimes (\mathcal{D}_1(t) \vee \mathcal{D}_2(t))$. Applying $a \otimes (b \vee c) = (a \otimes b) \vee (a \otimes c)$ to the latter expression, we get $\big((\mathcal{D}_1(t) \rightarrow \mathcal{D}'_1(t)) \wedge (\mathcal{D}_2(t) \rightarrow \mathcal{D}'_2(t))\big) \otimes (\mathcal{D}_1(t) \vee \mathcal{D}_2(t)) \leq \big((\mathcal{D}_1(t) \rightarrow \mathcal{D}'_1(t)) \otimes \mathcal{D}_1(t)\big) \vee \big((\mathcal{D}_2(t) \rightarrow \mathcal{D}'_2(t)) \otimes \mathcal{D}_2(t)\big)$. Using $a \otimes (a \rightarrow b) \leq b$ twice, it follows that $\big((\mathcal{D}_1(t) \rightarrow \mathcal{D}'_1(t)) \otimes \mathcal{D}_1(t)\big) \vee \big((\mathcal{D}_2(t) \rightarrow \mathcal{D}'_2(t)) \otimes \mathcal{D}_2(t)\big) \leq \mathcal{D}'_1(t) \vee \mathcal{D}'_2(t)$. Putting previous inequalities together, $\big(S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2)\big) \otimes (\mathcal{D}_1 \cup \mathcal{D}_2)(t) \leq (\mathcal{D}'_1 \cup \mathcal{D}'_2)(t)$ which proves (5). (6) can be proved analogously as (5); (7) can be proved analogously as (6) using monotony of $\otimes$; (8) follows from the fact that $a \rightarrow b \leq (c \rightarrow a) \rightarrow (c \rightarrow b)$. □

Using (4), we have the following consequence of Theorem 1:

Corollary 1. *For $\Diamond$ being $\cap$ and $\cup$, we have:*

$$E(\mathcal{D}_1, \mathcal{D}'_1) \wedge E(\mathcal{D}_2, \mathcal{D}'_2) \leq E(\mathcal{D}_1 \Diamond \mathcal{D}_2, \mathcal{D}'_1 \Diamond \mathcal{D}'_2). \tag{9}$$

$$E(\mathcal{D}_1, \mathcal{D}'_1) \otimes E(\mathcal{D}_2, \mathcal{D}'_2) \leq E(\mathcal{D}_1 \otimes \mathcal{D}_2, \mathcal{D}'_1 \otimes \mathcal{D}'_2). \tag{10}$$

$$E(\mathcal{D}_1, \mathcal{D}_2) \leq E(a \rightarrow \mathcal{D}_1, a \rightarrow \mathcal{D}_2). \tag{11}$$

Proof (sketch). For $\Diamond$ being $\cap$, (6) applied twice yields: $S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cap \mathcal{D}_2, \mathcal{D}'_1 \cap \mathcal{D}'_2)$ and $S(\mathcal{D}'_1, \mathcal{D}_1) \wedge S(\mathcal{D}'_2, \mathcal{D}_2) \leq S(\mathcal{D}'_1 \cap \mathcal{D}'_2, \mathcal{D}_1 \cap \mathcal{D}_2)$. Hence, (9) for $\cap$ follows using (2). The rest is analogous. □

Using the idea in the proof of Corollary 1, in order to prove that operation O preserves similarity, it suffices to check that O preserves (graded) subsethood. Thus, from now on, we shall only investigate whether operations preserve subsethood. In case of Cartesian products, we have:

Theorem 2. *Let $\mathcal{D}_1$ and $\mathcal{D}'_1$ be RDTs on relation scheme S and let $\mathcal{D}_2$ and $\mathcal{D}'_2$ be RDTs on relation scheme T such that $S \cap T = \emptyset$. Then,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \times \mathcal{D}_2, \mathcal{D}'_1 \times \mathcal{D}'_2), \tag{12}$$

Proof (sketch). The proof is analogous to that of (7). □

The following assertion shows that projection and similarity-based selection preserve subsethood degrees (and therefore similarities) of RDTs:

Table 2. Alternative ranks for houses for sale from Table 1

	agent	*id*	*sqft*	*age*	*location*	*price*
0.93	Brown	138	1185	48	Vestal	$228,500
0.91	Clark	140	1120	30	Endicott	$235,800
0.87	Brown	156	1300	85	Binghamton	$248,600
0.85	Brown	142	950	50	Binghamton	$189,000
0.82	Davis	189	1250	25	Binghamton	$287,300
0.79	Clark	158	1200	25	Vestal	$293,500
0.75	Davis	166	1040	50	Vestal	$286,200
0.37	Davis	112	1890	30	Endicott	$345,000

Theorem 3. *Let $\mathcal{D}$ and $\mathcal{D}'$ be RDTs on relation scheme T and let $y \in T$, $d \in D_y$, and $R \subseteq T$. Then,*

$$S(\mathcal{D}, \mathcal{D}') \leq S(\pi_R(\mathcal{D}), \pi_R(\mathcal{D}')), \tag{13}$$

$$S(\mathcal{D}, \mathcal{D}') \leq S(\sigma_{y \approx d}(\mathcal{D}), \sigma_{y \approx d}(\mathcal{D}')). \tag{14}$$

Proof (sketch). In oder to prove (13), we check $S(\mathcal{D}, \mathcal{D}') \otimes (\pi_R(\mathcal{D}))(r) \leq (\pi_R(\mathcal{D}'))(r)$ for any $r \in \mathrm{Tupl}(R)$. It means showing that

$$S(\mathcal{D}, \mathcal{D}') \otimes \textstyle\bigvee_{s \in \mathrm{Tupl}(T \setminus R)} \mathcal{D}(rs) \leq (\pi_R(\mathcal{D}'))(r).$$

Thus, is suffices to prove $S(\mathcal{D}, \mathcal{D}') \otimes \mathcal{D}(rs) \leq (\pi_R(\mathcal{D}'))(r)$ for all $s \in \mathrm{Tupl}(T \setminus R)$. Using monotony of $\otimes$, we get $S(\mathcal{D}, \mathcal{D}') \otimes \mathcal{D}(rs) \leq (\mathcal{D}(rs) \to \mathcal{D}'(rs)) \otimes \mathcal{D}(rs) \leq \mathcal{D}'(rs)$, because $rs \in \mathrm{Tupl}(T)$. Therefore, $S(\mathcal{D}, \mathcal{D}') \otimes \mathcal{D}(rs) \leq \mathcal{D}'(rs) \leq \bigvee_{s \in \mathrm{Tupl}(T \setminus R)} \mathcal{D}'(rs) = (\pi_R(\mathcal{D}'))(r)$, which proves the first claim of (13). In case of (14), we proceed analogously. □

Theorem 2 and Theorem 3 used together yield

Corollary 2. *Let $\mathcal{D}_1$ and $\mathcal{D}'_1$ be RDTs on relation scheme S and let $\mathcal{D}_2$ and $\mathcal{D}'_2$ be RDTs on relation scheme T such that $S \cap T = \emptyset$. Then,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \bowtie_{p \approx q} \mathcal{D}_2, \mathcal{D}'_1 \bowtie_{p \approx q} \mathcal{D}'_2). \tag{15}$$

for any $p \in S$ and $q \in T$ having the same domain with similarity. □

As a result, we have shown that important relational operations in our model (including similarity-based joins) preserve similarity defined by (2). Thus, we have provided a formal justification for the (intuitively expected but nontrivial) fact that similar input data yield similar results of queries.

Remark 3. In this paper, we have restricted ourselves only to a fragment of relational operations in our model. In [5], we have shown that in order to have a relational algebra whose expressive power is the same as the expressive power of the domain relational calculus, we have to consider additional operations of *residuum* (defined componentwise using $\to$) and *division*. Nevertheless, these two additional operations preserve E as well—it can be shown using similar arguments as in the proof of Theorem 1. As a consequence, the similarity is preserved by all queries that can be formulated in DRC [5].

4.2 Illustrative Example

Consider again the RDT from Table 1. The RDT can be seen as a result of querying a database of houses for sale where one wants to find a house which is sold for (approximately) `$200,000` and has (approximately) `1200` square feet. The attributes in the RDT are: real estate agent name (*agent*), house ID (*id*), square footage (*sqft*), house age (*age*), house location (*location*), and house price (*price*). In this example, the complete residuated lattice $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \to, 0, 1 \rangle$ serving as the structure of ranks will be the so-called Łukasiewicz algebra [2, 18, 19]. That is, $L = [0, 1]$, $\wedge$ and $\vee$ are minimum and maximum, respectively, and the multiplication and residuum are defined as follows: $a \otimes b = \max(a + b - 1, 0)$ and $a \to b = \min(1 - a + b, 1)$ for all $a, b \in L$.

Intuitively, it is natural to consider similarity of values in domains of *sqft*, *age*, *location*, and *price*. For instance, similarity of prices can be defined by $p_1 \approx_{price} p_2 = s(|p_2 - p_1|)$ using an antitone

scaling function $s\colon [0,\infty) \to [0,1]$ with $s(0) = 1$ (i.e., identical prices are fully similar). Analogously, a similarity of locations can be defined based on their geographical distance and/or based on their evaluation (safety, school districts, ...) by an expert. In contrast, there is no need to have similarities for *id* and *agents* because end-users do not look for houses based on (similarity of) their (internal) IDs which are kept as keys merely because of performance reasons. Obviously, there may be various reasonable similarity relations defined for the above-mentioned domains and their careful choice is an important task. In this paper, we neither explain nor recommend particular ways to do so because (i) we try to keep a general view of the problem and (ii) similarities on domains are purpose and user dependent.

Consider now the RDT in Table 2 defined over the same relation scheme as the RDT in Table 1. These two RDTs can be seen as two (slightly different) answers to the same query (when e.g., the domain similarities have been slightly changed) or answers to a modified query (e.g., "show all houses which are sold for (approximately) $210,000 and ..."). The similarity of both the RDTs given by (2) is 0.98 (very high). The results in the previous section say that if we perform any (arbitrarily complex) query (using the relational operations we consider in this paper) with Table 2 instead of Table 1, the results will be similar at least to degree 0.98.

Table 3. Join of Table 1 and the table of customers

	agent	*id*	*price*	*name*	*budget*
0.91	Brown	138	$228,500	Grant	$240,000
0.89	Brown	138	$228,500	Evans	$250,000
0.89	Brown	138	$228,500	Finch	$210,000
0.88	Clark	140	$235,800	Grant	$240,000
0.86	Clark	140	$235,800	Evans	$250,000
0.84	Brown	156	$248,600	Evans	$250,000
⋮	⋮	⋮	⋮	⋮	⋮
0.16	Davis	112	$345,000	Grant	$240,000
0.10	Davis	112	$345,000	Finch	$210,000

For illustration, consider an additional RDT of customers over relation scheme containing two attributes: *name* (customer name) and *budget* (price the customer is willing to pay for a house). In particular, let $\langle$Evans, $250,000$\rangle$, $\langle$Finch, $210,000$\rangle$, and $\langle$Grant, $240,000$\rangle$ be the only tuples in the RDT (all with ranks 1). The answer to the following query

$$\pi_{\{agent,id,price,name,budget\}}(\mathcal{D}_1 \bowtie_{price \approx budget} \mathcal{D}_c),$$

where $\mathcal{D}_1$ stands for Table 1 and $\mathcal{D}_c$ stands for the RDT of customers is in Table 3 (for brevity, some records are omitted). The RDT thus represents an answer to query "show deals for houses sold for (approximately) $200,000 with (approximately) 1200 square feet and customers so that their budget is similar to the house price". Furthermore, we can obtain an RDT of best agent-customer matching is we project the join onto *agent* and *name*:

$$\pi_{\{agent,name\}}(\mathcal{D}_1 \bowtie_{price \approx budget} \mathcal{D}_c).$$

The result of matching is in Table 4 (left). Due to our results, if we perform the same query with Table 2 instead of Table 1, the new result is guaranteed to be similar with the obtained result at least to degree 0.98. The result for Table 2 is shown in Table 4 (right).

4.3 Tuple-Based Similarity and Further Topics

While the rank-based similarity from Section 4.1 can be sufficient in many cases, there are situations where one wants to consider a similarity of RDTs based on ranks and (pairwise) similarity of tuples. For instance, if we take the RDT from Table 1 and make a new one by taking all tuples (keeping their ranks) and increasing the prices by one dollar, we will come up with an RDT which is, according to rank-based similarity, very different from the original one. Intuitively, one would expect to have a high degree of

Table 4. Results of agent-customer matching for Table 1 and Table 2

	agent	*name*
0.91	Brown	Grant
0.89	Brown	Evans
0.89	Brown	Finch
0.88	Clark	Grant
0.86	Clark	Evans
0.84	Clark	Finch
0.74	Davis	Evans
0.72	Davis	Grant
0.66	Davis	Finch

	agent	*name*
0.91	Brown	Grant
0.90	Clark	Grant
0.89	Brown	Evans
0.89	Brown	Finch
0.88	Clark	Evans
0.86	Clark	Finch
0.75	Davis	Evans
0.73	Davis	Grant
0.67	Davis	Finch

similarity of the RDTs because they differ only by a slight change in price. This issue can be solved by considering the following tuple-based degree of inclusion:

$$S^{\approx}(\mathcal{D}_1, \mathcal{D}_2) = \bigwedge_{t \in \mathrm{Tupl}(T)} \big(\mathcal{D}_1(t) \to \bigvee_{t' \in \mathrm{Tupl}(T)} \big(\mathcal{D}_2(t') \otimes t \approx t'\big)\big), \tag{16}$$

where $t \approx t' = \bigwedge_{y \in T} t(y) \approx_y t'(y)$ is a similarity of tuples t and t' over T, cf. [6]. In a similar way as in (4), we may define $E^{\approx}$ using $S^{\approx}$ instead of S.

Remark 4. By an easy inspection, $S(\mathcal{D}_1, \mathcal{D}_2) \leq S^{\approx}(\mathcal{D}_1, \mathcal{D}_2)$, i.e. (16) yields an estimate which is at least as high as (3) and analogously for E and $E^{\approx}$. Note that (16) has a natural meaning. Indeed, $S^{\approx}(\mathcal{D}_1, \mathcal{D}_2)$ can be understood as a degree to which the following statement is true: "If t belongs to $\mathcal{D}_1$, then there is t' which is similar to t and which belongs to $\mathcal{D}_2$". Hence, $E^{\approx}(\mathcal{D}_1, \mathcal{D}_2)$ is a degree to which for each tuple from $\mathcal{D}_1$ there is a similar tuple in $\mathcal{D}_2$ and *vice versa.* If $\mathbf{L}$ is a two-element Boolean algebra and each $\approx_y$ is an identity, then $E^{\approx}(\mathcal{D}_1, \mathcal{D}_2) = 1$ iff $\mathcal{D}_1$ and $\mathcal{D}_2$ are identical (in the usual sense).

For tuple-based inclusion (similarity) and for certain relational operations, we can prove analogous preservation formulas as in Section 4.1. For instance,

$$S^{\approx}(\mathcal{D}_1, \mathcal{D}_1') \wedge S(\mathcal{D}_2, \mathcal{D}_2') \leq S^{\approx}(\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{D}_1' \cup \mathcal{D}_2'), \tag{17}$$

$$S^{\approx}(\mathcal{D}_1, \mathcal{D}_1') \otimes S(\mathcal{D}_2, \mathcal{D}_2') \leq S^{\approx}(\mathcal{D}_1 \times \mathcal{D}_2, \mathcal{D}_1' \times \mathcal{D}_2'), \tag{18}$$

$$S^{\approx}(\mathcal{D}, \mathcal{D}') \leq S^{\approx}(\pi_R(\mathcal{D}), \pi_R(\mathcal{D}')). \tag{19}$$

On the other hand, similarity-based selection $\sigma_{y \approx d}$ (and, as a consequence, similarity-based join $\bowtie_{p \approx q}$) does not preserve $S^{\approx}$ in general which can be seen as a technical complication. This issue can be overcome by introducing a new type of selection $\sigma^{\approx}_{y \approx d}$ which is *compatible* with $S^{\approx}$. Namely, we can define

$$\big(\sigma^{\approx}_{y \approx d}(\mathcal{D})\big)(t) = \bigvee_{t' \in \mathrm{Tupl}(T)} \big(\mathcal{D}(t') \otimes t' \approx t \otimes t(y) \approx_y d\big). \tag{20}$$

For this notion, we can prove that $S^{\approx}(\mathcal{D}, \mathcal{D}') \leq S^{\approx}(\sigma^{\approx}_{y \approx d}(\mathcal{D}), \sigma^{\approx}_{y \approx d}(\mathcal{D}'))$. Similar extension can be done for any relational operation which does not preserve $S^{\approx}$ directly. Detailed description of the extension is postponed to a full version of the paper because of the limited scope.

4.4 Unifying Approach to Similarity of RDTs

In this section, we outline a general approach to similarity of RDTs that includes both the approaches from the previous sections. Interestingly, both (3) and (16) have a common generalization using truth-stressing hedges [19, 21]. Truth-stressing hedges represent unary operations on complete residuated lattices (denoted by $*$) that serve as interpretations of logical connectives like "very true", see [19]. Two boundary cases of hedges are (i) identity, i.e. $a^* = a$ ($a \in L$); (ii) globalization: $1^* = 1$, and $a^* = 0$ if $a < 1$. The globalization [31] is a hedge which can be interpreted as "fully true".

Let * be truth-stressing hedge on $\mathbf{L}$. For RDTs $\mathcal{D}_1, \mathcal{D}_2$ on T, we define the degree $S^{\approx}_*(\mathcal{D}_1, \mathcal{D}_2)$ of inclusion of $\mathcal{D}_1$ in $\mathcal{D}_2$ (with respect to *) by

$$S^{\approx}_*(\mathcal{D}_i, \mathcal{D}_j) = \bigwedge_{t \in \mathrm{Tupl}(T)} \big(\mathcal{D}_i(t) \to \bigvee_{t' \in \mathrm{Tupl}(T)} \big(\mathcal{D}_j(t') \otimes (t \approx t')^*\big)\big). \tag{21}$$

Now, it is easily seen that for $*$ being the identity, (21) coincides with (16); if $\approx$ is separating (i.e., $t_1 \approx t_2 = 1$ iff t_1 is identical to t_2) and $*$ is the globalization, (21) coincides with (3). Thus, both (3) and (16) are particular instances of (21) resulting by a choice of the hedge. Note that identity and globalization are two borderline cases of hedges. In general, complete residuated lattices admit other nontrivial hedges that can be used in (21). Therefore, the hedge in (21) serves as a parameter that has an influence on how much emphasis we put on the fact that two tuples are similar. In case of globalization, we put full emphasis, i.e., the tuples are required to be equal to degree 1 (exactly the same if $\approx$ is separating).

If we consider properties needed to prove analogous estimation formulas for general $S^{\approx}_{*}$ as we did in case of S and $S^{\approx}$, we come up with the following important property:

$$(r \approx s)^* \otimes (s \approx t)^* \leq (r \approx t)^*, \tag{22}$$

for every $r, s, t \in \mathrm{Tupl}(T)$ which can be seen as transitivity of $\approx$ with respect to $\otimes$ and *. Consider the following two cases in which (22) is satisfied:

Case 1: $*$ is globalization and $\approx$ is separating. If the left hand side of (22) is nonzero, then $r \approx s = 1$ and $s \approx t = 1$. Separability implies $r = s = t$, i.e. $(r \approx t)^* = 1^* = 1$, verifying (22).

Case 2: $\approx$ is transitive. In this case, since $a^* \otimes b^* \leq (a \otimes b)^*$ (follows from properties of hedges by standard arguments), transitivity of $\approx$ and monotony of * yield $(r \approx s)^* \otimes (s \approx t)^* \leq ((r \approx s) \otimes (s \approx t))^* \leq (r \approx t)^*$.

The following lemma shows that $S^{\approx}_{*}$ and consequently $E^{\approx}_{*}$ have properties that are considered natural for (degrees of) inclusion and similarity:

Lemma 1. *If $\approx$ satisfies* (22) *with respect to * then*

(*i*) *$S^{\approx}_{*}$ is a reflexive and transitive* **L***-relation, i.e. an* **L***-quasiorder.*

(*ii*) *$E^{\approx}_{*}$ defined by $E^{\approx}_{*}(\mathcal{D}_1, \mathcal{D}_2) = S^{\approx}_{*}(\mathcal{D}_1, \mathcal{D}_2) \wedge S^{\approx}_{*}(\mathcal{D}_2, \mathcal{D}_1)$ is a reflexive, symmetric, and transitive* **L***-relation, i.e. an* **L***-equivalence.*

Proof. The assertion follows from results in [2, Section 4.2] by taking into account that $\approx^*$ is reflexive, symmetric, and transitive with respect to $\otimes$. □

5 Conclusion and Future Research

We have shown that an important fragment of relational operation in similarity-based databases preserves various types of similarity. As a result, similarity of query results based on these relational operations can be estimated based on similarity of input data tables before the queries are executed. Furthermore, the results of this paper have shown a desirable important property of the underlying similarity-based model of data: slight changes in input data do not produce huge changes in query results. Future research will focus on the role of particular relational operations called similarity-based closures that play an important role in tuple-based similarities of RDTs. An outline of results in this direction is presented in [3].

References

1. S. Abiteboul *et al.* The Lowell database research self-assessment. *Communications of the ACM* 48(5):111-118, 2005.
2. R. Belohlavek. *Fuzzy Relational Systems: Foundations and Principles*. Kluwer, Academic/Plenum Publishers, New York, 2002.
3. R. Belohlavek, L. Urbanova, and V. Vychodil. Similarity of query results in similarity-based databases (*in preparation*).
4. R. Belohlavek and V. Vychodil. Logical foundations for similarity-based databases. *DASFAA 2009 Workshops*, LNCS 5667:137–151, 2009.
5. R. Belohlavek and V. Vychodil. Query systems in similarity-based databases: logical foundations, expressive power, and completeness. In: *Proc. ACM SAC* 2010, pp. 1648–1655.

6. R. Belohlavek and V. Vychodil. Codd's relational model from the point of view of fuzzy logic. *J. Logic and Computation* (to appear, doi: 10.1093/logcom/exp056).
7. G. Birkhoff: *Lattice theory*. First edition. American Mathematical Society, Providence, 1940.
8. B. P. Buckles and F. E. Petry. Fuzzy databases in the new era. ACM SAC 1995, pages 497–502, Nashville, TN, 1995.
9. R. Cavallo and M. Pittarelli. The theory of probabilistic databases. VLDB 1987, pp. 71–81.
10. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13(6):377–387, 1970.
11. N. Dalvi, C. Ré and D. Suciu. Probabilistic databases: diamonds in the dirt. *Communications of the ACM* 52:86–94, 2009.
12. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16:523–544, 2007.
13. N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. ACM PODS 2007, pp. 1–12.
14. C. J. Date. *Database Relational Model: A Retrospective Review and Analysis.* Addison Wesley, 2000.
15. R. Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record* 31(2):109–118, 2002.
16. G. Gerla. *Fuzzy Logic. Mathematical Tools for Approximate Reasoning.* Kluwer, Dordrecht, 2001.
17. J. A. Goguen. The logic of inexact concepts. *Synthese* 18:325–373, 1968–9.
18. S. Gottwald. Mathematical fuzzy logics. *Bulletin for Symbolic Logic* 14(2):210–239, 2008.
19. P. Hájek. *Metamathematics of Fuzzy Logic.* Kluwer, Dordrecht, 1998.
20. T. Imieliński, W. Lipski. Incomplete information in relational databases. *Journal of the ACM* 31:761–791, 1984.
21. P. Hájek. On very true. *Fuzzy Sets and Syst.* 124:329–333, 2001.
22. C. Koch. On query algebras for probabilistic databases. *SIGMOD Record* 37(4):78–85, 2008.
23. C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query Algebra and Optimization for Relational top-k queries. ACM SIGMOD 2005, pp. 131–142.
24. D. Maier. *The Theory of Relational Databases.* Comp. Sci. Press, Rockville, 1983.
25. D. Olteanu, J. Huang, C. Koch. Approximate confidence computation in probabilistic databases. *IEEE ICDE 2010*, pp. 145–156.
26. J. Pavelka: On fuzzy logic I, II, III. *Z. Math. Logik Grundlagen Math.* 25:45–52, 25:119–134, 25:447–464, 1979.
27. K. V. S. V. N. Raju and A. K. Majumdar. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *ACM Trans. Database Systems* Vol. 13, No. 2:129–166, 1988.
28. S. Shenoi and A. Melton. Proximity relations in the fuzzy relational database model. *Fuzzy Sets and Syst.* 100:51–62, 1999.
29. D. Suciu, D. Olteanu, C. Ré, C. Koch. *Probabilistic Databases.* Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2011.
30. Y. Takahashi. Fuzzy database query languages and their relational completeness theorem. *IEEE Trans. Knowledge and Data Engineering* 5:122–125, February 1993.
31. G. Takeuti and S. Titani. Globalization of intuitionistic set theory. *Annals of Pure and Applied Logic* 33: 195–211, 1987.

A Uniform Fixpoint Approach to the Implementation of Inference Methods for Deductive Databases

Andreas Behrend

University of Bonn,
Institute of Computer Science III,Römerstraße 164,
53117 Bonn, Germany
behrend@cs.uni-bonn.de

Abstract. Within the research area of deductive databases three different database tasks have been deeply investigated: query evaluation, update propagation and view updating. Over the last thirty years various inference mechanisms have been proposed for realizing these main functionalities of a rule-based system. However, these inference mechanisms have been rarely used in commercial DB systems until now. One important reason for this is the lack of a uniform approach well-suited for implementation in an SQL-based system. In this paper, we present such a uniform approach in form of a new version of the soft consequence operator. Additionally, we present improved transformation-based approaches to query optimization and update propagation and view updating which are all using this operator as underlying evaluation mechanism.

1 Introduction

The notion deductive database refers to systems capable of inferring new knowledge using rules. Within this research area, three main database tasks have been intensively studied: (recursive) query evaluation, update propagation and view updating. Despite of many proposals for efficiently performing these tasks, however, the corresponding methods have been implemented in commercial products (such as, e.g., Oracle or DB2) in a very limited way, so far. One important reason is that many proposals employ inference methods which are not directly suited for being transferred into the SQL world. For example, proof-based methods or instance-oriented model generation techniques (e.g. based on SLDNF) have been proposed as inference methods for view updating which are hardly compatible with the set-oriented bottom-up evaluation strategy of SQL.

In this paper, we present transformation-based methods to query optimization, update propagation and view updating which are well-suited for being transferred to SQL. Transformation-based approaches like Magic Sets [1] automatically transform a given database schema into a new one such that the evaluation of rules over the rewritten schema performs a certain database task more efficiently than with respect to the original schema. These approaches are well-suited for extending database systems, as new algorithmic ideas are solely incorporated into the transformation process, leaving the actual database engine with its own optimization techniques unchanged. In fact, rewriting techniques allow for implementing various database functionalities on the basis of one common inference engine. However, the application of transformation-based approaches with respect to stratifiable views [17] may lead to unstratifiable recursion within the rewritten schemata. Consequently, an elaborate and very expensive inference mechanism is generally required for their evaluation such as the alternating fixpoint computation or the residual program approach proposed by van Gelder [20] resp. Bry [10]. This is also the case for the kind of recursive views proposed by the SQL:1999 standard, as they cover the class of stratifiable views.

As an alternative, the soft consequence operator together with the soft stratification concept has been proposed by the author in [2] which allows for the efficient evaluation of Magic Sets transformed rules. This efficient inference method is applicable to query-driven as well as update-driven derivations. Query-driven inference is typically a top-down process whereas update-driven approaches are usually designed bottom-up. During the last 6 years, the idea of combining the advantages of top-down and bottom-up oriented inference has been consequently employed to enhance existing methods to query optimization [3] as well as update propagation [6] and to develop a new approach to view updating. In order to handle alternative derivations that may occur in view updating methods, an extended version of the original soft consequence

operator has to be developed. In this paper, this new version is presented, which is well-suited for efficiently determining the semantics of definite and indefinite databases but remains compatible with the set-oriented, bottom-up evaluation of SQL.

2 Basic concepts

A Datalog *rule* is a function-free clause of the form $H_1 \leftarrow L_1 \wedge \cdots \wedge L_m$ with $m \geq 1$ where H_1 is an atom denoting the rule's head, and $L_1, \ldots, L_m$ are literals, i.e. positive or negative atoms, representing its body. We assume all deductive rules to be *safe*, i.e., all variables occurring in the head or in any negated literal of a rule must be also present in a positive literal in its body. If $A \equiv p(t_1, \ldots, t_n)$ with $n \geq 0$ is a literal, we use $\texttt{vars}(A)$ to denote the set of variables occurring in A and $\texttt{pred}(A)$ to refer to the predicate symbol p of A. If A is the head of a given rule R, we use $\texttt{pred}(R)$ to refer to the predicate symbol of A. For a set of rules $\mathcal{R}$, $\texttt{pred}(\mathcal{R})$ is defined as $\cup_{r \in \mathcal{R}} \{\texttt{pred}(r)\}$. A *fact* is a ground atom in which every t_i is a constant.

A *deductive database* $\mathcal{D}$ is a triple $\langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ where $\mathcal{F}$ is a finite set of facts (called *base facts*), $\mathcal{I}$ is a finite set of integrity constraints (i.e.,positive ground atoms) and $\mathcal{R}$ a finite set of rules such that $\texttt{pred}(\mathcal{F}) \cap \texttt{pred}(\mathcal{R}) = \emptyset$ and $\texttt{pred}(\mathcal{I}) \subseteq \texttt{pred}(\mathcal{F} \cup \mathcal{R})$. Within a deductive database $\mathcal{D}$, a predicate symbol p is called derived (view predicate), if $p \in \texttt{pred}(\mathcal{R})$. The predicate p is called extensional (or base predicate), if $p \in \texttt{pred}(\mathcal{F})$. Let $\mathcal{H}_\mathcal{D}$ be the Herbrand base of $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$. The set of all derivable literals from $\mathcal{D}$ is defined as the well-founded model [21] for $(\mathcal{F} \cup \mathcal{R})$: $\mathcal{M}_\mathcal{D} := I^+ \cup \neg \cdot I^-$ where $I^+, I^- \subseteq \mathcal{H}_\mathcal{D}$ are sets of ground atoms and $\neg \cdot I^-$ includes all negations of atoms in I^-. The set I^+ represents the positive portion of the well-founded model while $\neg \cdot I^-$ comprises all negative conclusions. The semantics of a database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ is defined as the well-founded model $\mathcal{M}_\mathcal{D} := I^+ \cup \neg \cdot I^-$ for $\mathcal{F} \cup \mathcal{R}$ if all integrity constraints are satisfied in $\mathcal{M}_\mathcal{D}$, i.e., $\mathcal{I} \subseteq I^+$. Otherwise, the semantics of $\mathcal{D}$ is undefined. For the sake of simplicity of exposition, and without loss of generality, we assume that a predicate is either base or derived, but not both, which can be easily achieved by rewriting a given database.

Disjunctive Datalog extends Datalog by disjunctions of literals in facts as well as rule heads. A disjunctive Datalog rule is a function-free clause of the form $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n$ with $m, n \geq 1$ where the rule's head $A_1 \vee \ldots \vee A_m$ is a disjunction of positive atoms, and the rule's body $B_1, \ldots, B_n$ consists of literals, i.e. positive or negative atoms. A disjunctive fact $f \equiv f_1 \vee \ldots \vee f_k$ is a disjunction of ground atoms f_i with $i \geq 1$. f is called definite if $i = 1$. We solely consider stratifiable disjunctive rules only, that is, recursion through negative predicate occurrences is not permitted [17]. A stratification partitions a given rule set such that all positive derivations of relations can be determined before a negative literal with respect to one of those relations is evaluated. The semantics of a stratifiable disjunctive databases $\mathcal{D}$ is defined as the perfect model state $\mathcal{PM}_\mathcal{D}$ of $\mathcal{D}$ iff $\mathcal{D}$ is consistent [4, 11].

3 Transformation-Based Approaches

The need for a uniform inference mechanism in deductive databases is motivated by the fact that transformation-based approaches to query optimization, update propagation and view updating are still based on very different model generators. In this section, we briefly recall the state-of-the-art with respect to these transformation-based techniques by means of Magic Sets, Magic Updates and Magic View Updates. The last two approaches have been already proposed by the author in [6] and [7]. Note that we solely consider stratifiable rules for the given (external) schema. The transformed internal schema, however, may not always be stratifiable such that more general inference engines are required.

3.1 Query Optimization

Various methods for efficient bottom-up evaluation of queries against the intensional part of a database have been proposed, e.g. Magic Sets [1], Counting [9], Alexander method [19]). All these approaches are rewriting techniques for deductive rules with respect to a given query such that bottom-up materialization is performed in a goal-directed manner cutting down the number of irrelevant facts generated. In the following we will focus on Magic Sets as this approach has been accepted as a kind of standard in the field.

Magic Sets rewriting is a two-step transformation in which the first phase consists of constructing an adorned rule set, while the second phase consists of the actual Magic Sets rewriting. Within an adorned rule set, the predicate symbol of a literal is associated with an adornment which is a string consisting of letters `b` and `f`. While `b` represents a bound argument at the time when the literal is to be evaluated, `f` denotes a free argument. The adorned version of the deductive rules is constructed with respect to an adorned query and a selected sip strategy [18] which basically determines for each rule the order in which the body literals are to be evaluated and which bindings are passed on to the next literal. During the second phase of Magic Sets the adorned rules are rewritten such that bottom-up materialization of the resulting database simulates a top-down evaluation of the original query on the original database. For this purpose, each adorned rule is extended with a magic literal restricting the evaluation of the rule to the given binding in the adornment of the rule's head. The magic predicates themselves are defined by rules which define the set of relevant selection constants. The initial values corresponding to the query are given by the so-called magic seed. As an example, consider the following stratifiable rules $\mathcal{R}$

$$\mathtt{o}(\mathtt{X},\mathtt{Y}) \leftarrow \neg\mathtt{p}(\mathtt{Y},\mathtt{X}) \wedge \mathtt{p}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}(\mathtt{Z},\mathtt{Y})$$

and the query `?-o(1,2)` asking whether a path from node 1 to 2 exists but not vice versa. Assuming a full left-to-right sip strategy, Magic Sets yields the following deductive rules $\mathcal{R}_{ms}$

$$\mathtt{o_{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_o_{bb}}(\mathtt{X},\mathtt{Y}) \wedge \neg\mathtt{p_{bb}}(\mathtt{Y},\mathtt{X}) \wedge \mathtt{p_{bb}}(\mathtt{X},\mathtt{Y}) \qquad \mathtt{p_{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_p_{bb}}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{e}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{p_{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_p_{bb}}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{e}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p_{bb}}(\mathtt{Z},\mathtt{Y}) \qquad \mathtt{m\_p_{bb}}(\mathtt{Y},\mathtt{X}) \leftarrow \mathtt{m\_o_{bb}}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{m\_p_{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_o_{bb}}(\mathtt{X},\mathtt{Y}) \wedge \neg\mathtt{p_{bb}}(\mathtt{Y},\mathtt{X}) \qquad \mathtt{m\_o_{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_s\_o_{bb}}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{m\_p_{bb}}(\mathtt{Z},\mathtt{Y}) \leftarrow \mathtt{m\_p_{bb}}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{e}(\mathtt{X},\mathtt{Z})$$

as well as the magic seed fact $\mathtt{m\_s\_o_{bb}}(1,2)$. The Magic Sets transformation is sound for stratifiable databases. However, the resulting rule set may be no more stratifiable (as is the case in the above example) and more general approaches than iterated fixpoint computation are needed. For determining the well-founded model of general logic programs, the alternating fixpoint computation by Van Gelder [20] or the conditional fixpoint by Bry [10] could be used. The application of these methods, however, is not really efficient because the specific reason for the unstratifiability of the transformed rule sets is not taken into account. As an efficient alternative, the soft stratification concept together with the soft consequence operator [2] could be used for determining the positive part of the well-founded model (cf. Section 4).

3.2 Update Propagation

Determining the consequences of base relation changes is essential for maintaining materialized views as well as for efficiently checking integrity. Update propagation (UP) methods have been proposed aiming at the efficient computation of implicit changes of derived relations resulting from explicitly performed updates of extensional facts [13, 14, 16, 17]. We present a specific method for update propagation which fits well with the semantics of deductive databases and is based on the soft consequence operator again. We will use the notion *update* to denote the 'true' changes caused by a transaction only; that is, we solely consider sets of updates where compensation effects (i.e., given by an insertion and deletion of the same fact or the insertion of facts which already existed, for example) have already been taken into account.

The task of update propagation is to systematically compute the set of all induced modifications starting from the physical changes of base data. Technically, this is a set of delta facts for any affected relation which may be stored in corresponding delta relations. For each predicate symbol $p \in \mathtt{pred}(\mathcal{D})$, we will use a pair of delta relations $\langle \Delta_p^+, \Delta_p^- \rangle$ representing the insertions and deletions induced on p by an update on $\mathcal{D}$. The initial set of delta facts directly results from the given update and represents the so-called UP seeds. They form the starting point from which induced updates, represented by derived delta relations, are computed. In our transformation-based approach, so-called *propagation rules* are employed for computing delta relations. A propagation rule refers to at least one delta relation in its body in order to provide a focus on the underlying changes when computing induced updates. For showing the effectiveness of an induced update, however, references to the state of a relation before and after the base update has been performed

are necessary. As an example of this propagation approach, consider again the rules for relation p from Subsection 3.1. The UP rules $\mathcal{R}^{\Delta}$ with respect to insertions into e are as follows :

$$\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})$$
$$\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}^{\mathtt{new}}(\mathtt{Z},\mathtt{Y}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})$$
$$\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z},\mathtt{Y}) \wedge \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Z}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})$$

For each relation p we use p^{old} to refer to its old state before the changes given in the delta relations have been applied whereas p^{new} is used to refer to the new state of p. These state relations are never completely computed but are queried with bindings from the delta relation in the propagation rule body and thus act as a test of effectiveness. In the following, we assume the old database state to be present such that the adornment *old* can be omitted. For simulating the new database state from a given update so called $transition\ rules$ [16] are used. The transition rules $\mathcal{R}^{\Delta}_{\tau}$ for simulating the required new states of e and p are:

$$\mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}) \wedge \neg \Delta^-_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) \qquad \mathtt{p}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) \qquad \mathtt{p}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}^{\mathtt{new}}(\mathtt{Z},\mathtt{Y})$$

Note that the new state definition of intensional predicates only indirectly refers to the given update in contrast to extensional predicates. If $\mathcal{R}$ is stratifiable, the rule set $\mathcal{R} \cup \mathcal{R}^{\Delta} \cup \mathcal{R}^{\Delta}_{\tau}$ will be stratifiable, too (cf. [6]). As $\mathcal{R} \cup \mathcal{R}^{\Delta} \cup \mathcal{R}^{\Delta}_{\tau}$ remains to be stratifiable, iterated fixpoint computation could be employed for determining the semantics of these rules and the induced updates defined by them. However, all state relations are completely determined which leads to a very inefficient propagation process. The reason is that the supposed evaluation over the two consecutive database states is performed using deductive rules which are not specialized with respect to the particular updates that are propagated. This weakness of propagation rules in view of a bottom-up materialization will be cured by incorporating Magic Sets.

Magic Updates

The aim is to develop an UP approach which is automatically limited to the affected delta relations. The evaluation of side literals and effectiveness tests is restricted to the updates currently propagated. We use the Magic Sets approach for incorporating a top-down evaluation strategy by considering the currently propagated updates in the dynamic body literals as abstract queries on the remainder of the respective propagation rule bodies. Evaluating these propagation queries has the advantage that the respective state relations will only be partially materialized. As an example, let us consider the specific deductive database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ with $\mathcal{R}$ consisting of the well-known rules for the transitive closure p of relation e:

$\underline{\mathcal{R}}$: $\mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y})$
$\mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Z}), \mathtt{p}(\mathtt{Z},\mathtt{Y})$

$\underline{\mathcal{F}}$: `edge(1,2), edge(1,4), edge(3,4)`
`edge(10,11), edge(11,12), ..., edge(98,99), edge(99,100)`

Note that the derived relation p consists of 4098 tuples. Suppose a given update contains the new tuple $e(2,3)$ to be inserted into $\mathcal{D}$ and we are interested in finding the resulting consequences for p. Computing the induced update by evaluating the stratifiable propagation and transition rules would lead to the generation of 94 new state facts for relation e, 4098 old state facts for p and $4098 + 3$ new state facts for p. The entire number of generated facts is 8296 for computing the three induced insertions $\Delta^+_{\mathtt{p}}(1,3), \Delta^+_{\mathtt{p}}(2,3), \Delta^+_{\mathtt{p}}(2,4)\}$ with respect to p.

However, the application of the Magic Updates rewriting with respect to the propagation queries $\{\Delta^+_{\mathtt{p}}(\mathtt{Z},\mathtt{Y}), \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y}), \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Z})\}$ provides a much better focus on the changes to e. Within its application, the following subquery rules

$$\mathtt{m\_p}^{\mathtt{new}}_{\mathtt{bf}}(\mathtt{Z}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Z}) \qquad \mathtt{m\_p}_{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{m\_e}^{\mathtt{new}}_{\mathtt{fb}}(\mathtt{Z}) \leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z},\mathtt{Y}) \qquad \mathtt{m\_p}_{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}^{\mathtt{new}}_{\mathtt{bf}}(\mathtt{Z},\mathtt{Y})$$
$$\mathtt{m\_p}_{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z},\mathtt{Y}) \wedge \mathtt{e}^{\mathtt{new}}_{\mathtt{fb}}(\mathtt{X},\mathtt{Z})$$

are generated. The respective queries $Q = \{m\_e_{fb}^{new}, m\_p_{bf}^{new}, \ldots\}$ allow to specialize the employed transition rules, e.g.

$$\mathtt{e}_{\mathtt{fb}}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_e}_{\mathtt{fb}}^{\mathtt{new}}(\mathtt{Y}) \wedge \mathtt{e}(\mathtt{X},\mathtt{Y}) \wedge \neg \Delta_{\mathtt{e}}^{-}(\mathtt{X},\mathtt{Y})$$
$$\mathtt{e}_{\mathtt{fb}}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{m\_e}_{\mathtt{fb}}^{\mathtt{new}}(\mathtt{Y}) \wedge \Delta_{\mathtt{e}}^{+}(\mathtt{X},\mathtt{Y})$$

such that only relevant state tuples are generated. We denote the Magic Updates transformed rules $\mathcal{R} \cup \mathcal{R}^{\Delta} \cup \mathcal{R}_{\tau}^{\Delta}$ by $\mathcal{R}_{mu}^{\Delta}$. Despite of the large number of rules in $\mathcal{R}_{mu}^{\Delta}$, the number of derived results remains relatively small. Quite similar to the Magic sets approach, the Magic Updates rewriting may result in an unstratifiable rule set. This is also the case for our example where the following negative cycle occurs in the respective dependency graph:

$$\Delta_{\mathtt{p}}^{+} \xrightarrow{pos} \mathtt{m\_p}_{\mathtt{bb}} \xrightarrow{pos} \mathtt{p}_{\mathtt{bb}} \xrightarrow{neg} \Delta_{\mathtt{p}}^{+}$$

In [6] it has been shown, however, that the resulting rules must be at least softly stratifiable such that the soft consequence operator could be used for efficiently computing their well-founded model. Computing the induced update by evaluating the Magic Updates transformed rules leads to the generation of two new state facts for e, one old state fact and one new state fact for p. The entire number of generated facts is 19 in contrast to 8296 for computing the three induced insertions with respect to p.

3.3 View Updates

Bearing in mind the numerous benefits of the afore mentioned methods to query optimization and update propagation, it seemed worthwhile to develop a similar, i.e., incremental and transformation-based, approach to the dual problem of view updating. In contrast to update propagation, view updating aims at determining one or more base relation updates such that all given update requests with respect to derived relations are satisfied after the base updates have been successfully applied. In the following, we recall a transformation-based approach to incrementally compute such base updates for stratifiable databases proposed by the author in [7]. The approach extends and integrates standard techniques for efficient query answering, integrity checking and update propagation. The analysis of view updating requests usually leads to alternative view update realizations which are represented in disjunctive form.

Magic View Updates

In our transformation-based approach, true view updates (VU) are considered only, i.e., ground atoms which are presently not derivable for atoms to be inserted, or are derivable for atoms to be deleted, respectively. A method for view updating determines sets of alternative updates (called VU realization) satisfying a given request. There may be infinitely many realizations and even realizations of infinite size which satisfy a given VU request. In our approach, a breadth-first search is employed for determining a set of minimal realizations. A realization is minimal in the sense that none of its updates can be removed without losing the property of being a realization. As each level of the search tree is completely explored, the result usually consists of more than one realization. If only VU realizations of infinite size exist, our method will not terminate.

Given a VU request, view updating methods usually determine subsequent VU requests in order to find relevant base updates. Similar to delta relations for UP we will use the notion *VU relation* to access individual view updates with respect to the relations of our system. For each relation $p \in \mathtt{pred}(\mathcal{R} \cup \mathcal{F})$ we use the VU relation $\nabla_{\mathtt{p}}^{+}(\vec{x})$ for tuples to be inserted into $\mathcal{D}$ and $\nabla_{\mathtt{p}}^{-}(\vec{x})$ for tuples to be deleted from $\mathcal{D}$. The initial set of delta facts resulting from a given VU request is again represented by so-called VU seeds. Starting from the seeds, so-called VU rules are employed for finding subsequent VU requests systematically. These rules perform a top-down analysis in a similar way as the bottom-up analysis implemented by the UP rules. As an example, consider the following database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ with $\mathcal{F} = \{r_2(2), s(2)\}$, $\mathcal{I} = \{ic(2)\}$ and the rules $\mathcal{R}$:

$$\mathtt{p}(\mathtt{X}) \leftarrow \mathtt{q}_1(\mathtt{X}) \qquad \mathtt{q}_1(\mathtt{X}) \leftarrow \mathtt{r}_1(\mathtt{X}) \wedge \mathtt{s}(\mathtt{X})$$
$$\mathtt{p}(\mathtt{X}) \leftarrow \mathtt{q}_2(\mathtt{X}) \qquad \mathtt{q}_2(\mathtt{X}) \leftarrow \mathtt{r}_2(\mathtt{X}) \wedge \neg \mathtt{s}(\mathtt{X})$$
$$\mathtt{ic}(2) \leftarrow \neg \mathtt{au}(2) \qquad \mathtt{au}(\mathtt{X}) \leftarrow \mathtt{q}_2(\mathtt{X}) \wedge \neg \mathtt{q}_1(\mathtt{X})$$

The corresponding set of VU rules $\mathcal{R}^{\nabla}$ with respect to $\nabla^+_{\mathtt{p}}(2)$ is given by:

$$\nabla^+_{\mathtt{q}_1}(\mathtt{X}) \vee \nabla^+_{\mathtt{q}_1}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{p}}(\mathtt{X})$$
$$\nabla^+_{\mathtt{r}_1}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q}_1}(\mathtt{X}) \wedge \neg \mathtt{r}_1(\mathtt{X}) \qquad \nabla^+_{\mathtt{r}_2}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q}_2}(\mathtt{X}) \wedge \neg \mathtt{r}_2(\mathtt{X})$$
$$\nabla^+_{\mathtt{s}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q}_1}(\mathtt{X}) \wedge \neg \mathtt{s}(\mathtt{X}) \qquad \nabla^-_{\mathtt{s}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q}_2}(\mathtt{X}) \wedge \mathtt{s}(\mathtt{X})$$

In contrast to the UP rules from Section 3.2, no explicit references to the new database state are included in the above VU rules. The reason is that these rules are applied iteratively over several intermediate database states before the minimal set of realizations has been found. Hence, the apparent references to the old state really refer to the current state which is continuously modified while computing VU realizations. These predicates solely act as tests again queried with respect to bindings from VU relations and thus will never be completely evaluated.

Evaluating these rules using model generation with disjunctive facts leads to two alternative updates, insertion $\{r_1(2)\}$ and deletion $\{s(2)\}$, induced by the derived disjunction $\nabla^+_{\mathtt{r}_1}(2) \vee \nabla^-_{\mathtt{s}}(2)$. Obviously, the second update represented by $\nabla^-_{\mathtt{s}}(2)$ would lead to an undesired side effect by means of an integrity violation. In order to provide a complete method, however, such erroneous/incomplete paths must be also explored and side effects repaired if possible. Determining whether a computed update will lead to a consistent database state or not can be done by applying a bottom-up UP process at the end of the top-down phase leading to an irreparable constraint violation with respect to $\nabla^-_s(2)$:

$$\nabla^-_{\mathtt{s}}(2) \Rightarrow \Delta^+_{\mathtt{q}_2}(2) \Rightarrow \Delta^+_{\mathtt{p}}(2), \Delta^+_{\mathtt{au}}(2) \Rightarrow \Delta^-_{\mathtt{ic}}(2) \rightsquigarrow false$$

In order to see whether the violated constraint can be repaired, the subsequent view update request $\nabla^+_{\mathtt{ic}}(2)$ with respect to $\mathcal{D}$ ought to be answered. The application of $\mathcal{R}^{\nabla}$ yields

$$\begin{array}{l} \qquad\qquad\quad \Rightarrow \nabla^-_{\mathtt{q}_2}(2), \nabla^+_{\mathtt{q}_2}(2) \rightsquigarrow false \\ \nabla^+_{\mathtt{ic}}(2) \Rightarrow \nabla^-_{\mathtt{aux}}(2) \Updownarrow \\ \qquad\qquad\quad \Rightarrow \nabla^+_{\mathtt{q}_1}(2) \Rightarrow \nabla^+_{\mathtt{s}}(2), \nabla^-_{\mathtt{s}}(2) \rightsquigarrow false \end{array}$$

showing that this request cannot be satisfied as inconsistent subsequent view update requests are generated on this path. Such erroneous derivation paths will be indicated by the keyword $false$. The reduced set of updates - each of them leading to a consistent database state only - represents the set of realizations $\Delta^+_{\mathtt{r}_1}(2)$.

An induced deletion of an integrity constraint predicate can be seen as a side effect of an 'erroneous' VU. Similar side effects, however, can be also found when induced changes to the database caused by a VU request may include derived facts which had been actually used for deriving this view update. This effect is shown in the following example for a deductive database $\mathcal{D} = \langle \mathcal{R}, \mathcal{F}, \mathcal{I} \rangle$ with $\mathcal{R} = \{\mathtt{h}(\mathtt{X}) \leftarrow \mathtt{p}(\mathtt{X}) \wedge \mathtt{q}(\mathtt{X}) \wedge \mathtt{i}, \mathtt{i} \leftarrow \mathtt{p}(\mathtt{X}) \wedge \neg \mathtt{q}(\mathtt{X})\}$, $\mathcal{F} = \{\mathtt{p}(1)\}$, and $\mathcal{I} = \emptyset$. Given the VU request $\nabla^+_{\mathtt{h}}(1)$, the overall evaluation scheme for determining the only realization $\{\Delta^+_{\mathtt{q}}(1), \Delta^+_{\mathtt{p}}(c^{new_1})\}$ would be as follows:

$$\begin{array}{l} \qquad\qquad\qquad\qquad\quad \Rightarrow \nabla^+_{\mathtt{p}}(\mathtt{c}^{\mathtt{new}_1}) \\ \nabla^+_{\mathtt{h}}(1) \Rightarrow \nabla^+_{\mathtt{q}}(1) \Rightarrow \Delta^+_{\mathtt{q}}(1) \Rightarrow \Delta^-_{\mathtt{i}} \Rightarrow \nabla^+_{\mathtt{i}} \Updownarrow \\ \qquad\qquad\qquad\qquad\quad \Rightarrow \nabla^-_{\mathtt{q}}(1), \nabla^+_{\mathtt{q}}(1) \rightsquigarrow false \end{array}$$

The example shows the necessity of compensating side effects, i.e., the compensation of the 'deletion' Δ^-_i (that prevents the 'insertion' $\Delta^+_h(1)$) caused by the tuple $\nabla^+_q(1)$. In general the compensation of side effects, however, may in turn cause additional side effects which have to be 'repaired'. Thus, the view updating method must alternate between top-down and bottom-up phases until all possibilities for compensating side effects (including integrity constraint violations) have been considered, or a solution has been found. To this end, so-called *VU transition rules* $\mathcal{R}^{\nabla}_{\tau}$ are used for restarting the VU analysis. For example, the compensation of violated integrity constraints can be realized by using the following kind of transition rule $\Delta^-_{ic}(\vec{c}) \rightarrow \nabla^+_{ic}(\vec{c})$ for each ground literal $ic(\vec{c}) \in \mathcal{I}$. VU transition rules make sure that erroneous solutions are evaluated to $false$ and side effects are repaired.

Having the rules for the direct and indirect consequences of a given VU request, a general application scheme for systematically determining VU realizations can be defined (see[7] for details). Instead of using simple propagation rules $\mathcal{R} \uplus \mathcal{R}^{\Delta} \uplus \mathcal{R}^{\Delta}_{\tau}$, however, it is much more efficient to employ the corresponding Magic Update rules. The top-down analysis rules $\mathcal{R} \uplus \mathcal{R}^{\nabla}$ and the bottom-up consequence

analysis rules $\mathcal{R}^{\Delta}_{mu} \uplus \mathcal{R}^{\nabla}_{\tau}$ are alternating applied. Note that the disjunctive rules $\mathcal{R} \uplus \mathcal{R}^{\nabla}$ are stratifiable while $\mathcal{R}^{\Delta}_{mu} \uplus \mathcal{R}^{\nabla}_{\tau}$ is softly stratifiable such that a perfect model state [4, 11] and a well-founded model generation must alternately be applied. The iteration stops as soon as a realization for the given VU request has been found. The correctness of this approach has been already shown in [7].

4 Consequence Operators and Fixpoint Computations

In the following, we summarize the most important fixpoint-based approaches for definite as well as indefinite rules. All these methods employ so-called consequence operators which formalize the application of deductive rules for deriving new data. Based on their properties, a new uniform consequence operator is developed subsequently.

4.1 Definite Rules

First, we recall the iterated fixpoint method for constructing the well-founded model of a stratifiable database which coincides with its perfect model [17].

Definition 1. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, λ a stratification on $\mathcal{D}$, $\mathcal{R}_1 \uplus \ldots \uplus \mathcal{R}_n$ the partition of $\mathcal{R}$ induced by λ, $I \subseteq \mathcal{H}_{\mathcal{D}}$ a set of ground atoms, and $[[\mathcal{R}]]_I$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I. Then we define*

1. *the immediate consequence operator $T_{\mathcal{R}}(I)$ as*

$$T_{\mathcal{R}}(I) := \{H \mid \begin{array}{l} H \in I \vee \exists r \in [[\mathcal{R}]]_I : r \equiv H \leftarrow L_1 \wedge \ldots \wedge L_n \\ \text{such that } L_i \in I \text{ for all positive literals } L_i \\ \text{and } L \notin I \text{ for all negative literals } L_j \equiv \neg L\}, \end{array}$$

2. *the iterated fixpoint M_n as the last Herbrand model of the sequence*

$$M_1 := \texttt{lfp}\ (T_{\mathcal{R}_1}, \mathcal{F}),\ M_2 := \texttt{lfp}\ (T_{\mathcal{R}_2}, M_1),\ \ldots,\ M_n := \texttt{lfp}\ (T_{\mathcal{R}_n}, M_{n-1}),$$

   where $\texttt{lfp}\ (T_{\mathcal{R}}, \mathcal{F})$ denotes the least fixpoint of operator $T_{\mathcal{R}}$ containing $\mathcal{F}$.

3. *and the iterated fixpoint model $\mathcal{M}^i_{\mathcal{D}}$ as*

$$\mathcal{M}^i_{\mathcal{D}} := M_n \uplus \neg \cdot \overline{M_n}.$$

This constructive definition of the iterated fixpoint model is based on the immediate consequence operator introduced by van Emden and Kowalski. In [17] it has been shown that the perfect model of a stratifiable database $\mathcal{D}$ is identical with the iterated fixpoint model $\mathcal{M}^i_{\mathcal{D}}$ of $\mathcal{D}$.

Stratifiable rules represent the most important class of deductive rules as they cover the expressiveness of recursion in SQL:1999. Our transformation-based approaches, however, may internally lead to unstratifiable rules for which a more general inference method is necessary. In case that unstratifiability is caused by the application of Magic Sets, the so-called soft stratification approach proposed by the author in [2] could be used.

Definition 2. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, λ^s a soft stratification on $\mathcal{D}$, $\mathcal{P} = P_1 \uplus \ldots \uplus P_n$ the partition of $\mathcal{R}$ induced by λ^s, and $I \subseteq \mathcal{H}_{\mathcal{D}}$ a set of ground atoms. Then we define*

1. *the soft consequence operator $T^s_{\mathcal{P}}(I)$ as*

$$T^s_{\mathcal{P}}(I) := \begin{cases} I & if\ T_{P_j}(I) = I\ forall\ j \in \{1, \ldots, n\} \\ T_{P_i}(I) & with\ i\ =\ \texttt{min}\{j \mid T_{P_j}(I) \supsetneq I\},\ otherwise. \end{cases}$$

   where T_{P_i} denotes the immediate consequence operator.

2. *and the soft fixpoint model $\mathcal{M}^s_{\mathcal{D}}$ as*

$$\mathcal{M}^s_{\mathcal{D}} := \texttt{lfp}\ (T^s_{\mathcal{P}}, \mathcal{F}) \uplus \neg \cdot \overline{(\texttt{lfp}\ (T^s_{\mathcal{P}}, \mathcal{F}))}.$$

Note that the soft consequence operator is based upon the immediate consequence operator and can even be used to determine the iterated fixpoint model of a stratifiable database [6]. As an even more general alternative, the alternating fixpoint model for arbitrary unstratifiable rules has been proposed in [12] on the basis of the eventual consequence operator.

Definition 3. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, $I^+, I^- \subseteq \mathcal{H}_\mathcal{D}$ sets of ground atoms, and $[[\mathcal{R}]]_{I^+}$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I^+. Then we define*

1. *the eventual consequence operator $\widehat{T}_\mathcal{R}\langle I^- \rangle$ as*

$$\widehat{T}_\mathcal{R}\langle I^- \rangle(I^+) := \{H \mid \begin{array}{l} H \in I^+ \vee \exists r \in [[\mathcal{R}]]_{I^+} : r \equiv H \leftarrow L_1 \wedge \ldots \wedge L_n \\ \textit{such that } L_i \in I^+ \textit{ for all positive literals } L_i \\ \textit{and } L \notin I^- \textit{ for all negative literals } L_j \equiv \neg L\},\end{array}$$

2. *the eventual consequence transformation $\widehat{S}_\mathcal{D}$ as*

$$\widehat{S}_\mathcal{D}(I^-) := \texttt{lfp}(\widehat{T}_\mathcal{R}\langle I^- \rangle, \mathcal{F}),$$

3. *and the alternating fixpoint model $\mathcal{M}^a_\mathcal{D}$ as*

$$\mathcal{M}^a_\mathcal{D} := \texttt{lfp}\,(\widehat{S}^2_\mathcal{D}, \emptyset) \uplus \neg \cdot \overline{\widehat{S}^2_\mathcal{D}(\texttt{lfp}\,(\widehat{S}^2_\mathcal{D}, \emptyset))}\,,$$

where $\widehat{S}^2_\mathcal{D}$ denotes the nested application of the eventual consequence transformation, i.e., $\widehat{S}^2_\mathcal{D}(I^-) = \widehat{S}_\mathcal{D}(\widehat{S}_\mathcal{D}(I^-))$.

In [12] it has been shown that the alternating fixpoint model $\mathcal{M}^a_\mathcal{D}$ coincides with the well-founded model of a given database $\mathcal{D}$. The induced fixpoint computation may indeed serve as a universal model generator for arbitrary classes of deductive rules. However, the eventual consequence operator is computationally expensive due to the intermediate determination of supersets of sets of true atoms. With respect to the discussed transformation-based approaches, the iterated fixpoint model could be used for determining the semantics of the stratifiable subset of rules in $\mathcal{R}_{ms}$ for query optimization, $\mathcal{R}^\Delta_{mu}$ for update propagation, and $\mathcal{R}^\Delta_{mu} \uplus \mathcal{R}^\nabla_\tau$ for view updating. If these rule sets contain unstratifiable rules, the soft or alternating fixpoint model generator ought be used while the first has proven to be more efficient than the latter [2]. None of the above mentioned consequence operators, however, can deal with indefinite rules necessary for evaluating the view updating rules $\mathcal{R} \uplus \mathcal{R}^\nabla$.

4.2 Indefinite Rules

In [4], the author proposed a consequence operator for the efficient bottom-up state generation of stratifiable disjunctive deductive databases. To this end, a new version of the immediate consequence operator based on hyperresolution has been introduced which extends Minker's operator for positive disjunctive Datalog rules [15]. In contrast to already existing model generation methods our approach for efficiently computing perfect models is based on state generation. Within this disjunctive consequence operator, the mapping `red` on indefinite facts is employed which returns non-redundant and subsumption-free representations of disjunctive facts. Additionally, the mapping $\texttt{min\_models}(F)$ is used for determining the set of minimal Herbrand models from a given set of disjunctive facts F. We identify a disjunctive fact with a set of atoms such that the occurrence of a ground atom A within a fact f can also be written as $A \in f$. The set difference operator can then be used to remove certain atoms from a disjunction while the empty set as result is interpreted as $false$.

Definition 4. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable disjunctive database rules,λ a stratification on $\mathcal{D}$, $\mathcal{R}_1 \uplus \ldots \uplus \mathcal{R}_n$ the partition of $\mathcal{R}$ induced by λ, I an arbitrary subset of indefinite facts from the disjunctive Herbrand base [11] of $\mathcal{D}$, and $[[\mathcal{R}]]_I$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I Then we define.*

1. *the disjunctive consequence operator $T^{state}_\mathcal{R}$ as*

$$T_{\mathcal{R}}^{state}(I) := \texttt{red}(\{H \mid H \in I \vee \exists r \in [[\mathcal{R}]]_I : r \equiv A_1 \vee \ldots \vee A_l \leftarrow L_1 \wedge \ldots \wedge L_n$$
$$\textit{with } H = (A_1 \vee \cdots \vee A_l \vee f_1 \setminus L_1 \vee \cdots \vee f_n \setminus L_n \vee C)$$
$$\textit{such that } f_i \in I \wedge L_i \in f_i \textit{ for all positive literals } L_i$$
$$\textit{and } L_j \notin I \textit{ for all negative literals } L_j \equiv \neg L$$
$$\textit{and } (L_j \in C \Leftrightarrow \exists \mathcal{M} \in \texttt{min\_models}(I) :$$
$$L_j \in \mathcal{M} \textit{ for at least one negative literal } L_j$$
$$\textit{and } L_k \in \mathcal{M} \textit{ for all positive literals } L_k$$
$$\textit{and } A_l \notin \mathcal{M} \textit{ for all head literals of } r)\})$$

2. *the iterated fixpoint state S_n as the last minimal model state of the sequence*

   $S_1 := \texttt{lfp}\ (T_{\mathcal{R}_1}^{state}, \mathcal{F})$, $S_2 := \texttt{lfp}\ (T_{\mathcal{R}_2}^{state}, S_1)$, …, $S_n := \texttt{lfp}\ (T_{\mathcal{R}_n}^{state}, S_{n-1})$,

3. *and the iterated fixpoint state model $\mathcal{MS}_{\mathcal{D}}$ as*

   $\mathcal{MS}_{\mathcal{D}} := S_n \uplus \neg \cdot \overline{S_n}$.

In [4] it has been shown that the iterated fixpoint state model $\mathcal{MS}_{\mathcal{D}}$ of a disjunctive database $\mathcal{D}$ coincides with the perfect model state of $\mathcal{D}$. It induces a constructive method for determining the semantics of stratifiable disjunctive databases. The only remaining question is how integrity constraints are handled in the context of disjunctive databases. We consider again definite facts as integrity constraints, only, which must be derivable in every model of the disjunctive database. Thus, only those models from the iterated fixpoint state are selected in which the respective definite facts are derivable. To this end, the already introduced keyword *false* can be used for indicating and removing inconsistent model states. The database is called consistent iff at least one consistent model state exists.

This proposed inference method is well-suited for determining the semantics of stratifiable disjunctive databases with integrity constraints. And thus, it seems to be suited as the basic inference mechanism for evaluating view updating rules. The problem is, however, that the respective rules contain unstratifiable definite rules which cannot be evaluated using the inference method proposed above. Hence, the evaluation techniques for definite (Section 4.1) and indefinite rules (Section 4.2) do not really fit together and a new uniform approach is needed.

5 A Uniform Fixpoint Approach

In this section, a new version of the soft consequence operator is proposed which is suited as efficient state generator for softly stratifiable definite as well as stratifiable indefinite databases. The original version of the soft consequence operator $T_{\mathcal{P}}^{s}$ is based on the immediate consequence operator by van Emden and Kowalski and can be applied to an arbitrary partition $\mathcal{P}$ of a given set of definite rules. Consequently, its application does not always lead to correct derivations. In fact, this operator has been designed for the application to softly stratified rules resulting from the application of Magic Sets. However, this operator is also suited for determining the perfect model of a stratifiable database.

Lemma 1. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable database and λ a stratification of $\mathcal{R}$ inducing the partition $\mathcal{P}$ of $\mathcal{R}$. The perfect model $\mathcal{M}_{\mathcal{D}}$ of $\langle \mathcal{F}, \mathcal{R} \rangle$ is identical with the soft fixpoint model of $\mathcal{D}$, i.e.,*

$$\mathcal{M}_{\mathcal{D}} = \texttt{lfp}(T_{\mathcal{P}}^{s}, \mathcal{F}) \uplus \neg \cdot \overline{\texttt{lfp}(T_{\mathcal{P}}^{s}, \mathcal{F})}.$$

Proof. This property follows from the fact that for every partition $\mathcal{P} = P_1 \uplus \ldots P_n$ induced by a stratification, the condition $\texttt{pred}(P_i) \cap \texttt{pred}(P_j) = \emptyset$ with $i \neq j$ must necessarily hold. As soon as the application of the immediate consequence operator T_{P_i} with respect to a certain layer P_i generates no new facts anymore, the rules in P_i can never fire again. The application of the incorporated `min` function then induces the same sequence of Herbrand models as in the case of the iterated fixpoint computation. □

Another property we need for extending the original soft consequence operator is about the application of T^{state} to definite rules and facts.

Lemma 2. *Let r be an arbitrary definite rule and f be a set of arbitrary definite facts. The single application of r to f using the immediate consequence operator or the disjunctive consequence operator, always yields the same result, i.e.,*

$$T_r(f) \;=\; T_r^{state}(f).$$

Proof. The proof follows from the fact that all non-minimal conclusions of T^{state} are immediately eliminated by the subsumption operator `red`. □

The above proposition establishes the relationship between the definite and indefinite case showing that the disjunctive consequence operator represents a generalization of the immediate one. Thus, its application to definite rules and facts can be used to realize the same derivation process as the one performed by using the immediate consequence operator. Based on the two properties from above, we can now consistently extend the definition of the soft consequence operator which allows its application to indefinite rules and facts, too.

Definition 5. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be an arbitrary disjunctive database, I an arbitrary subset of indefinite facts from the disjunctive Herbrand base of $\mathcal{D}$, and $\mathcal{P} = P_1 \cup \ldots \cup P_n$ a partition of $\mathcal{R}$. The general soft consequence operator $T_{\mathcal{P}}^g(I)$ is defined as*

$$T_{\mathcal{P}}^g(I) := \begin{cases} I & if\ T_{P_j}(I) = I\ forall\ j \in \{1,\ldots,n\} \\ T_{P_i}^{state}(I) & with\ i \;=\; \mathtt{min}\{j \mid T_{P_j}^{state}(I) \supsetneq I\},\ otherwise. \end{cases}$$

where $T_{P_i}^{state}$ denotes the disjunctive consequence operator.

In contrast to the original definition, the general soft consequence operator is based on the disjunctive operator $T_{P_i}^{state}$ instead of the immediate consequence operator. The least fixpoint of $T_{\mathcal{P}}^g$ can be used to determine the perfect model of definite as well as indefinite stratifiable databases and the well-founded model of softly stratifiable definite databases.

Theorem 1 *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable disjunctive database and λ a stratification of $\mathcal{R}$ inducing the partition $\mathcal{P}$ of $\mathcal{R}$. The perfect model state $\mathcal{PS}_{\mathcal{D}}$ of $\langle \mathcal{F}, \mathcal{R} \rangle$ is identical with the least fixpoint model of $T_{\mathcal{P}}^g$, i.e.,*

$$\mathcal{PS}_{\mathcal{D}} = \mathtt{lfp}(T_{\mathcal{P}}^g, \mathcal{F}) \cup \neg \cdot \overline{\mathtt{lfp}(T_{\mathcal{P}}^g, \mathcal{F})}.$$

Proof. The proof directly follows from the correctness of the fixpoint computations for each stratum as shown in [4] and the same structural argument already used in Lemma 1. □

The definition of $\mathtt{lfp}(T_{\mathcal{P}}^g, \mathcal{F})$ induces a constructive method for determining the perfect model state as well as the well-founded model of a given database. Thus, it forms a suitable basis for the evaluation of the rules $\mathcal{R}_{ms}$ for query optimization, $\mathcal{R}_{mu}^{\Delta}$ for update propagation, and $\mathcal{R}_{mu}^{\Delta} \cup \mathcal{R}_{\tau}^{\nabla}$ as well as $\mathcal{R} \cup \mathcal{R}^{\nabla}$ for view updating. This general approach to defining the semantics of different classes of deductive rules is surprisingly simple and induces a rather efficient inference mechanism in contrast to general well-founded model generators. The soft stratification concept, however, is not yet applicable to indefinite databases because ordinary Magic Sets can not be used for indefinite clauses. Nevertheless, the resulting extended version of the soft consequence operator can be used as a uniform basis for the evaluation of all transformation-based techniques mentioned in this paper.

6 Conclusion

In this paper, we have presented an extended version of the soft consequence operator for the efficient top-down and bottom-up reasoning in deductive databases. This operator allows for the efficient evaluation of softly stratifiable incremental expressions and stratifiable disjunctive rules. It solely represents a theoretical approach but provides insights into design decisions for extending the inference component of commercial database systems. The relevance and quality of the transformation-based approaches, however, has been already shown in various practical research projects (e.g. [5, 8]) at the University of Bonn.

References

1. BANCILHON, F., RAMAKRISHNAN, R.: *An Amateur's Introduction to Recursive Query Processing Strategies*. SIGMOD Conference 1986: 16-52.
2. BEHREND, A.: *Soft stratification for magic set based query evaluation in deductive databases.* PODS 2003, New York, June 9–12, pages 102-110.
3. BEHREND, A.: *Optimizing Exitstential Queries in Stratifiable Deductive Databases.* SAC 2005: 623-628.
4. BEHREND, A.: *A Fixpoint Approach to State Generation for Stratifiable Disjunctive Databases.* ADBIS 2007: 283-297
5. BEHREND, A., DORAU, C., MANTHEY, R., SCHÜLLER, G.:*Incremental view-based analysis of stock market data streams.* IDEAS 2008, pages 269–275, ACM, 2008.
6. BEHREND, A., MANTHEY, R.: *Update Propagation in Deductive Databases Using Soft Stratification.* ADBIS 2004: 22-36
7. BEHREND, A., MANTHEY R.: *A Transformation-Based Aproach to View Updating in Stratifiable Deductive Databases.* FOIKS 2008: 253-271
8. BEHREND, A., SCHÜLLER, G., MANTHEY, R.: *AIMS: An Sql-Based System for Airspace Monitoring.* IWGS 2010, pages 31–38, ACM.
9. BEERI, C., RAMAKRISHNAN, R.: *On the Power of Magic*. JLP 10(1/2/3&4): 255-299 (1991).
10. BRY, F.: *Logic Programming as Constructivism: A Formalization and its Application to Databases*. PODS 1989: 34-50.
11. FERNANDEZ, J. A., MINKER, J.: *Semantics of Disjunctive Deductive Databases.* ICDT 1992, volume 646 of LNCS, pages 21–50, Springer.
12. KEMP, D., SRIVASTAVA, D., STUCKEY, P.: *Bottom-Up Evaluation and Query Optimization of Well-Founded Models*. TCS 146(1 & 2): 145-184 (1995).
13. KÜCHENHOFF, V.: *On the efficient computation of the difference between consecutive database states.* DOOD 1991, volume 566 of LNCS, pages 478–502, December 1991, Springer.
14. MANTHEY, R.: *Reflections on some fundamental issues of rule-based incremental update propagation.* DAISD 1994: 255-276, September 19-21, Universitat Politècnica de Catalunya.
15. MINKER, J.:*On Indefinite Databases and the Closed World Assumption*. CADE 1982: 292-308.
16. OLIVÉ, A.: *Integrity constraints checking in deductive databases.* VLDB 1991, pages 513–523.
17. PRZYMUSINSKI, T. C.: *On the Declarative Semantics of Deductive Databases and Logic Programs*. Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988, pages 193-216.
18. RAMAKRISHNAN, R.: *Magic Templates: A Spellbinding Approach to Logic Programs*. JLP 11(3&4): 189-216 (1991).
19. ROHMER, J., LESCOEUR, R., KERISIT, J.-M.: *The Alexander Method - A Technique for the Processing of Recursive Axioms in Deductive Databases*. New Generation Computing 4(3): 273-285 (1986).
20. VAN GELDER, A.: *The alternating fixpoint of logic programs with negation.* Journal of Computer and System Sciences, 47(1):185–221, August 1993.
21. VAN GELDER, A., ROSS, K. A., SCHLIPF, J. S.: *The Well-Founded Semantics for General Logic Programs*. Journal of the ACM 38(3): 620-650 (1991).

dynPARTIX - A Dynamic Programming Reasoner for Abstract Argumentation*

Wolfgang Dvořák, Michael Morak, Clemens Nopp, and Stefan Woltran

Institute of Information Systems
Vienna University of Technology, Austria

Abstract. The aim of this paper is to announce the release of a novel system for abstract argumentation which is based on decomposition and dynamic programming. We provide first experimental evaluations to show the feasibility of this approach.

1 Introduction

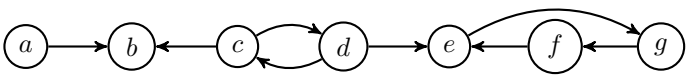
Argumentation has evolved as an important field in AI, with abstract argumentation frameworks (AFs, for short) as introduced by Dung [4] being its most popular formalization. Several semantics for AFs have been proposed (see e.g. [2] for an overview), but here we shall focus on the so-called preferred semantics. Reasoning under this semantics is known to be intractable [5]. An interesting approach to dealing with intractable problems comes from parameterized complexity theory which suggests to focus on parameters that allow for fast evaluations as long as these parameters are kept small. One important parameter for graphs (and thus for argumentation frameworks) is tree-width, which measures the "tree-likeness" of a graph. To be more specific, tree-width is defined via a certain decomposition of graphs, the so-called tree decomposition. Recent work [6] describes novel algorithms for reasoning in the preferred semantics, such that the performance mainly depends on the tree-width of the given AF, but the running times remain linear in the size of the AF. To put this approach to practice, we shall use the *SHARP* framework[1], a C++ environment which includes heuristic methods to obtain tree decompositions [3], provides an interface to run algorithms on these decompositions, and offers further useful features, for instance for parsing the input. For a description of the *SHARP* framework, see [8].

The main purpose of our work here is to support the theoretical results from [6] with experimental ones. Therefore we use different classes of AFs and analyze the performance of our approach compared to an implementation based on answer-set programming (see [7]). Our prototype system together with the used benchmark instances is available as a ready-to-use tool from `http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/`.

2 Background

Argumentation Frameworks. An *argumentation framework (AF)* is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation. If $(a, b) \in R$ we say a attacks b. An $a \in A$ is *defended* by a set $S \subseteq A$ iff for each $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$. An AF can naturally be represented as a digraph.

Example 1. Consider the AF $F = (A, R)$, with $A = \{a, b, c, d, e, f, g\}$ and $R = \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, g), (f, e), (g, f)\}$. The graph representation of F is given as follows:



* Supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-028, by the Austrian Science Fund (FWF) under grant P20704-N18, and by the Vienna University of Technology program "Innovative Ideas".

[1] `http://www.dbai.tuwien.ac.at/research/project/sharp`

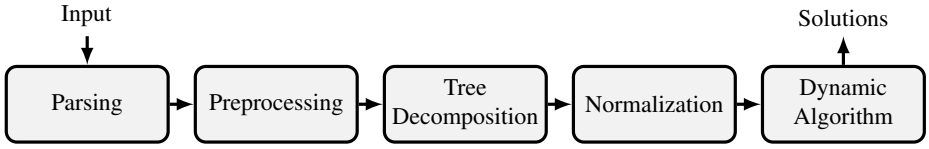


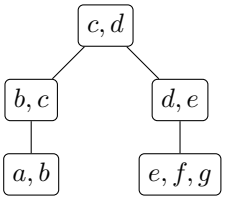
Fig. 1. Architecture of the *SHARP* framework.

We require the following semantical concepts: Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is (i) *conflict-free* in F, if there are no $a, b \in S$, such that $(a, b) \in R$; (ii) *admissible* in F, if S is conflict-free in F and each $a \in S$ is defended by S; (iii) a *preferred extension* of F, if S is a $\subseteq$-maximal admissible set in F. For the AF in Example 1, we get the admissible sets $\{\}, \{a\}, \{c\}, \{d\}, \{d, g\}, \{a, c\}, \{a, d\}$, and $\{a, d, g\}$. Consequently, the preferred extensions of this framework are $\{a, c\}, \{a, d, g\}$.

The typical reasoning problems associated with AFs are the following: (1) Credulous acceptance asks whether a given argument is contained in at least one preferred extension of a given AF; (2) skeptical acceptance asks whether a given argument is contained in all preferred extensions of a given AF. Credulous acceptance is NP-complete, while skeptical acceptance is even harder, namely Π_2^P-complete [5].

Tree Decompositions and Tree-width. As already outlined, tree decompositions will underlie our implemented algorithms. We briefly recall this concept (which is easily adapted to AFs). A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree and $\mathcal{X} = (X_t)_{t \in V_\mathcal{T}}$ is a set of so-called bags, which has to satisfy the following conditions: (a) $\bigcup_{t \in V_\mathcal{T}} X_t = V$, i.e. $\mathcal{X}$ is a cover of V; (b) for each $v \in V$, $\mathcal{T}|_{\{t | v \in X_t\}}$ is connected; (c) for each $\{v_i, v_j\} \in E$, $\{v_i, v_j\} \subseteq X_t$ for some $t \in V_\mathcal{T}$. The width of a tree decomposition is given by $\max\{|X_t| \mid t \in V_\mathcal{T}\} - 1$. The *tree-width* of G is the minimum width over all tree decompositions of G.

It can be shown that our example AF has tree-width 2 and next we illustrate a tree decomposition of width 2:



Dynamic programming algorithms traverse such tree decompositions (for our purposes we shall use so-called normalized decompositions, however) and compute local solutions for each node in the decomposition. Thus the combinatorial explosion is now limited to the size of the bags, that is, to the width of the given tree decomposition. For the formal definition of the algorithms, we refer to [6].

3 Implementation and *SHARP* Framework

dynPARTIX implements these algorithms using the *SHARP* framework [8], which is a purpose-built framework for implementing algorithms that are based on tree decompositions. Figure 1 shows the typical architecture, that systems working with the *SHARP* framework follow. In fact, *SHARP* provides interfaces and helper methods for the Preprocessing and Dynamic Algorithm steps as well as ready-to-use implementations of various tree decomposition heuristics, i.e. Minimum-Fill, Maximum-Cardinality-Search and Minimum-Degree heuristics (cf. [3]).

dynPARTIX builds on normalized tree decompositions provided by *SHARP*, which contain four types of nodes: Leaf-, Branch-, Introduction- and Removal-nodes. To implement our algorithms we just have to provide the methods and data structures for each of these node types (see [6] for the formal details). In short, the tree decomposition is traversed in a bottom-up manner, where at each node a table of all possible

partial solutions is computed. Depending on the node type, it is then modified accordingly and passed on to the respective parent node. Finally one can obtain the complete solutions from the root node's table.

SHARP handles data-flow management and provides data structures where the calculated (partial) solutions to the problem under consideration can be stored. The amount of dedicated code for *dynPARTIX* comes to around 2700 lines in C++. Together with the *SHARP* framework (and the used libraries for the tree-decomposition heuristics), our system roughly comprises of 13 000 lines of C++ code.

4 System Specifics

Currently the implementation is able to calculate the admissible and preferred extensions of the given argumentation framework and to check if credulous or skeptical acceptance holds for a specified argument. The basic usage of *dynPARTIX* is as follows:

```
> ./dynpartix [-f <file>] [-s <semantics>]
         [--enum | --count | --cred <arg> | --skept <arg>]
```

The argument `-f <file>` specifies the input file, the argument `-s <semantics>` selects the semantics to reason with, i.e. either admissible or preferred, and the remaining arguments choose one of the reasoning modes.

Input file conventions: We borrow the input format from the *ASPARTIX* system [7]. *dynPARTIX* thus handles text files where an argument a is encoded as `arg(a)` and an attack (a, b) is encoded as `att(a,b)`. For instance, consider the following encoding of our running example and let us assume that it is stored in a file `inputAF`.

```
arg(a). arg(b). arg(c). arg(d). arg(e). arg(f). arg(g).
att(a,b). att(c,b). att(c,d). att(d,c).
att(d,e). att(e,g). att(f,e). att(g,f).
```

Enumerating extensions: First of all, *dynPARTIX* can be used to compute extensions, i.e. admissible sets and preferred extensions. For instance to compute the admissible sets of our running example one can use the following command:

```
> ./dynpartix -f inputAF -s admissible
```

Credulous Reasoning: dynPARTIX decides credulous acceptance using proof procedures for admissible sets (even if one reasons with preferred semantics) to avoid unnecessary computational costs. The following statement decides if the argument d is credulously accepted in our running example.

```
> ./dynpartix -f inputAF -s preferred --cred d
```

Indeed the answer would be *YES* as $\{a, d, g\}$ is a preferred extension.

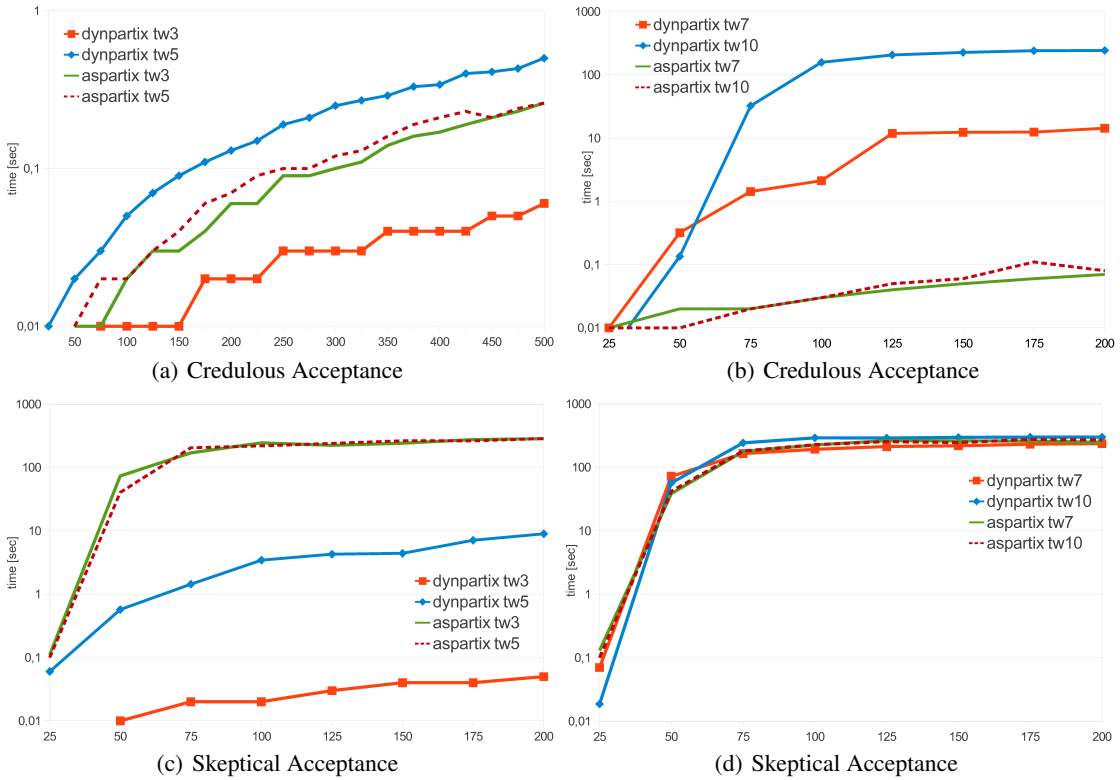
Skeptical Reasoning: To decide skeptical acceptance, *dynPARTIX* uses proof procedures for preferred extensions which usually results in higher computational costs (but is unavoidable due to complexity results). To decide if the argument d is skeptically accepted, the following command is used:

```
> ./dynpartix -f inputAF -s preferred --skept d
```

Here the answer would be *NO* as $\{a, c\}$ is a preferred extension not containing d.

Counting Extensions: Recently the problem of counting extensions has gained some interest [1]. We note that our algorithms allow counting without an explicit enumeration of all extensions (thanks to the particular nature of dynamic programming; see also [9]). Counting preferred extensions with *dynPARTIX* is done by

```
> ./dynpartix -f inputAF -s preferred --count
```



(a) Credulous Acceptance (b) Credulous Acceptance

(c) Skeptical Acceptance (d) Skeptical Acceptance

Fig. 2. Runtime behaviour of *dynPARTIX* for graphs of different tree-width compared with the *ASPARTIX* system.

5 Benchmark Tests

In this section we compare *dynPARTIX* with *ASPARTIX* [7], one of the most efficient reasoning tools for abstract argumentation (for an overview of existing argumentation systems see [7]). For our benchmarks we used randomly generated AFs of low tree-width. To ensure that AFs are of a certain tree-width we considered random grid-structured AFs. In such a grid-structured AF each argument is arranged in an $n \times m$ grid and attacks are only allowed between neighbours in the grid (we used a 8-neighborhood here to allow odd-length cycles). When generating the instances we varied the following parameters: the number of arguments; the tree-width; and the probability that an possible attack is actually in the AF.

The benchmark tests were executed on an Intel®Core™2 CPU 6300@1.86GHz machine running SUSE Linux version 2.6.27.48. We generated a total of 4800 argumentation frameworks with varying parameters as mentioned above. The corresponding runtimes are illustrated in Figure 2. The two graphs on the left-hand side compare the running times of *dynPARTIX* and *ASPARTIX* (using dlv) on instances of small treewidth (viz. 3 and 5). For the graphs on the right-hand side, we have used instances of higher width. Results for credulous acceptance are given in the upper graphs and those for skeptical acceptance in the lower graphs. The y-axis gives the runtimes in logarithmic scale; the x-axis shows the number of arguments. Note that the upper-left picture has different ranges on the axes compared to the three other graphs. We remark that the test script stopped a calculation if it was not finished after 300 seconds. For these cases we stored the value of 300 seconds in the database.

Interpretation of the Benchmark Results: We observe that, independent of the reasoning mode, the runtime of *ASPARTIX* is only minorly affected by the tree-width while *dynPARTIX* strongly benefits from a low tree-width, as expected by theoretical results [6].

For the *credulous acceptance* problem we have that our current implementation is competitive only up to tree-width 5. This is basically because *ASPARTIX* is quite good at this task. Considering Figures 2(a) and 2(b), there is to note that for credulous acceptance *ASPARTIX* decided every instance in less than 300 seconds, while *dynPARTIX* exceeded this value in 4% of the cases.

Now let us consider the *skeptical acceptance* problem. As mentioned before, skeptical acceptance is much harder computationally than credulous acceptance, which is reflected by the bad runtime behaviour of *ASPARTIX*. Indeed we have that for tree-width ≤ 5, *dynPARTIX* has a significantly better runtime behaviour, and that it is competitive on the whole set of test instances. As an additional comment to Figures 2(c) and 2(d), we note that for skeptical acceptance, *dynPARTIX* was able to decide about 71% of the test cases within the time limit, while *ASPARTIX* only finished 41%.

Finally let us briefly mention the problem of *Counting preferred extensions*. On the one side we have that *ASPARTIX* has no option for explicit counting extensions, so the best thing one can do is enumerating extensions and then counting them. It can easily be seen that this can be quite inefficient, which is reflected by the fact that *ASPARTIX* only finished 21% of the test instances in time. On the other hand we have that the dynamic algorithms for counting preferred extensions and deciding skeptical acceptance are essentially the same and thus have the same runtime behaviour.

6 Future work

We identify several directions for future work. First, a more comprehensive empirical evaluation would be of high value. For instance, it would be interesting to explore how our algorithms perform on real world instances. To this end, we need more knowledge about the tree-width typical argumentation instances comprise, i.e. whether it is the case that such instances have low tree-width. Due to the unavailability of benchmark libraries for argumentation, so far we had to omit such considerations.

Second, we see the following directions for further development of *dynPARTIX*: Enriching the framework with additional argumentation semantics mentioned in [2]; implementing further reasoning modes, which can be efficiently computed on tree decompositions, e.g. ideal reasoning; and optimizing the algorithms to benefit from recent developments in the SHARP framework.

References

1. Pietro Baroni, Paul E. Dunne, and Massimiliano Giacomin. On extension counting problems in argumentation frameworks. In Proc. *COMMA 2010*, pages 63–74, 2010
2. Pietro Baroni and Massimiliano Giacomin. Semantics of abstract argument systems. In *Argumentation in Artificial Intelligence*, pages 25–44. Springer, 2009.
3. Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, pages 1–11, 2008.
4. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
5. Paul E. Dunne and Trevor J. M. Bench-Capon. Coherence in finite argument systems. *Artif. Intell.*, 141(1/2):187–203, 2002.
6. Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for argumentation. In Proc. *KR'10*, pages 112–122, 2010.
7. Uwe Egly, Sarah Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *In Argument and Computation*, 1(2):147–177, 2010.
8. Michael Morak. SHARP - A smart hypertree-decomposition-based algorithm framework for parameterized problems. Technische Universität Wien, 2010.
   `http://www.dbai.tuwien.ac.at/research/project/sharp/sharp.pdf`
9. Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. of Discrete Algorithms*, 8(1): 50–64, 2010.

Nested HEX-Programs*

Thomas Eiter, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,tkren,redl}@kr.tuwien.ac.at

Abstract. Answer-Set Programming (ASP) is an established declarative programming paradigm. However, classical ASP lacks subprogram calls as in procedural programming, and access to external computations (like remote procedure calls) in general. The feature is desired for increasing modularity and—assuming proper access in place—(meta-)reasoning over subprogram results. While HEX-programs extend classical ASP with external source access, they do not support calls of (sub-)programs upfront. We present *nested* HEX-*programs*, which extend HEX-programs to serve the desired feature, in a user-friendly manner. Notably, the answer sets of called sub-programs can be individually accessed. This is particularly useful for applications that need to reason over answer sets like belief set merging, user-defined aggregate functions, or preferences of answer sets.

1 Introduction

Answer-Set Programming, based on [8], has been established as an important declarative programming formalism [3]. However, a shortcoming of classical ASP is the lack of means for modular programming, i.e., dividing programs into several interacting components. Even though reasoners such as DLV, CLASP, and DLVHEX allow to partition programs into several files, they are still viewed as a single monolithic sets of rules. On top of that, passing input to selected (sub-)programs is not possible upfront.

In procedural programming, the idea of calling subprograms and processing their output is in permanent use. Also in functional programming such modularity is popular. This helps reducing development time (e.g., by using third-party libraries), the length of source code, and, last but not least, makes code human-readable. Reading, understanding, and debugging a typical size application written in a monolithic program is cumbersome. Modular extensions of ASP have been considered [9, 5] with the aim of building an overall answer set from program modules; however, multiple results of subprograms (as typical for ASP) are respected, and no reasoning about such results is supported. XASP [11] is an SMODELS interface for XSB-Prolog. This system is related to our work, but in this scenario the meta-reasoner is Prolog and thus different from the semantics of its subprograms, which are under stable model semantics. The subprograms are monolithic programs and cannot make further calls. This is insufficient for some applications, e.g., for the MELD belief set merging system, which require hierarchical nesting of arbitrary depth. Adding such nesting to available approaches is not easily possible and requires to adapt systems similar to our approach.

HEX-programs [6] extend ASP with higher-order atoms, which allow the use of predicate variables, and external atoms, through which external sources of computation can be accessed. But HEX-programs do not support modularity and meta-reasoning directly. In this context, modularity means the encapsulation of subprograms which interact through well-defined interfaces only, and meta-reasoning requires reasoning over *sets of* answer sets. Moreover, in HEX-programs external sources are realized as procedural C++ functions. Therefore, as soon as external sources are queried, we leave the declarative formalism. However, the generic notion of external atom, which facilitates a bidirectional data flow between the logic program and an external source (viewed as abstract Boolean function), can be utilized to provide these features.

To this end, we present *nested* HEX-*programs*, which support (possibly parameterized) *subprogram calls*. It is the nature of nested hex-programs to have multiple HEX-programs which reason over the answer sets of each individual subprogram. This can be done in a user-friendly way and enables the user to write purely *declarative* applications consisting of multiple interacting modules. Notably, call results and answer

* This research has been supported by the Austrian Science Fund (FWF) project P20840 and P20841, and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

sets are *objects* that can be accessed by identifiers and processed in the calling program. Thus, different from [9, 5] and related formalisms, this enables *(meta)-reasoning about the set of answer sets* of a program. In contrast to [11], both the calling and the called program are in the same formalism. In particular, the calling program has also a multi-model semantics. As an important difference to [1], nested HEX-programs do not require extending the syntax and semantics of the underlying formalism, which is the HEX-semantics. The integration is, instead, by defining some external atoms (which is already possible in ordinary HEX-programs), making the approach simple and user-friendly for many applications. Furthermore, as nested HEX-programs are based on HEX-programs, they additionally provide access to external sources other than logic programs. This makes nested HEX-programs a powerful formalism, which has been implemented using the DLVHEX reasoner for HEX-programs; applications like belief set merging [10] show its potential and usefulness.

2 HEX-Programs

We briefly recall HEX-programs, which have been introduced in [6] as a generalization of (disjunctive) extended logic programs under the answer set semantics [8]; for more details and background, we refer to [6]. A HEX-program consists of rules of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ ,$$

where each a_i is a classical literal, i.e., an atom $p(t_1, \ldots, t_l)$ or a negated atom $\neg p(t_1, \ldots, t_l)$, and each b_j is either a classical literal or an external atom, and not is negation by failure (under stable semantics). An *external atom* is of the form

$$\&g[q_1, \ldots, q_k](t_1, \ldots, t_l) \ ,$$

where g is an external predicate name, the q_i are predicate names or constants, and the t_j are terms. Informally, the semantics of an external g is given by a $k + l + 1$-ary Boolean *oracle function* $f_{\&g}$. The external atom is true relative to an interpretation I and a grounding substitution θ iff $f_{\&g}(I, q_1, \ldots, q_k, t_1\theta, \ldots, t_l\theta) = 1$. Via such atoms, arbitrary (computable) functions can be included. E.g., built-in functions can be realized via external atoms, or library functions such as string manipulations, sorting routines, etc. As external sources need not be on the same machine, knowledge access across the Web is possible, e.g., belief set import. Strictly, [6] omits classical negation $\neg$ but the extension is routine; furthermore, [6] also allows terms for the q_i and variables for predicate names, which we do not consider.

Example 1. Suppose an external knowledge base consists of an RDF file located on the web at http://.../data.rdf. Using an external atom $\&rdf[<url>](X, Y, Z)$, we may access all RDF triples (s, p, o) at the URL specified with $<url>$. To form belief sets of pairs that drop the third argument from RDF triples, we may use the rule

$$bel(X, Y) \leftarrow \&rdf[\text{http://.../data.rdf}](X, Y, Z) \ .$$

The semantics of HEX-program is given via answer sets, which are sets of ground literals closed under the rules that satisfy a stability condition as in [8]; we refer to [6] for technical details. The above program has a single answer set which consists of all literal $bel(c_1, c_2)$ such some RDF triple (c_1, c_2, c_3) occurs at the respective URL.

We use the DLVHEX system from http://www.kr.tuwien.ac.at/research/systems/dlvhex/ as a backend. DLVHEX implements (a fragment of) HEX-programs. It provides a plugin mechanism for external atoms. Besides library atoms, the user can defined her own atoms, where for evaluation a C++ routine must be provided.

3 Nested HEX-Programs

Limitations of ASP. As a simple example demonstrating the limits of ordinary ASP, assume a program computing the shortest paths between two (fixed) nodes in a connected graph. The answer sets of this program then correspond to the shortest paths. Suppose we are just interested in the *number* of such paths. In a procedural setting, this is easily computed: if a function returns all these paths in an array, linked list, or similar data structure, then counting its elements is trivial.

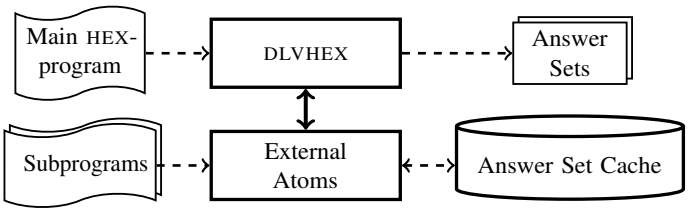


Fig. 1: System Architecture of Nestex HEX (data flow ⇢, control flow →)

In ASP, the solution is non-trivial if the given program must not be modified (e.g., if it is provided by a third party); above, we must count the answer sets. Thus, we need to reason on *sets of* answer sets, which is infeasible inside the program. Means to call the program at hand and reason about the results of this *"callee"* (*subprogram*) in the *"calling program"* (*host program*) would be useful. Aiming at a logical counterpart to procedural function calls, we define a framework which allows to input facts to the subprogram prior to its execution. Host and subprograms are decoupled and interact merely by relational input and output values. To realize this mechanism, we exploit external atoms, leading to nested HEX-programs.

Architecture. Nested HEX-programs are realized as a plugin for the reasoner DLVHEX,[1] which consists of a *set of external atoms* and an *answer cache* for the results of subprograms (see Fig. 1). Technically, the implementation is part of the belief set merging system MELD, which is an application on top of a nested HEX-programs core. This core can be used independently from the rest of the system.

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered in the host program, the plugin creates another instance of the reasoner to evaluate the subprogram. Its result is then stored in the answer cache and identified with a unique *handle*, which can later be used to reference the result and access its components (e.g., predicate names, literals, arguments) via other special external atoms.

There are two possible sources for the called subprogram: (1) either it is *directly embedded* in the host program, or (2) it is *stored in a separate file*. In (1), the rules of the subprogram must be represented within the host program. To this end, they are encoded as string constants. An embedded program must not be confused with a subset of the rules of the host program. Even though it is syntactically part of it, it is logically separated to allow independent evaluation. In (2) merely the *path* to the location of the external program in the file system is given. Compared to embedded subprograms, code can be reused without the need to copy, which is clearly advantageous when the subprogram changes. We now present concrete external atoms $\&callhex_n$, $\&callhexfile_n$, $\&answersets$, $\&predicates$, and $\&arguments$.

External Atoms for Subprogram Handling. We start with two families of external atoms

$$\&callhex_n[\mathtt{P}, p_1, \ldots, p_n](H) \quad \text{and} \quad \&callhexfile_n[\mathtt{FN}, p_1, \ldots, p_n](H)$$

that allow to execute a subprogram given by a string `P` respectively in a file `FN`; here n is an integer specifying the number of predicate names p_i, $1 \leq i \leq n$, used to define the input facts. When evaluating such an external atom relative to an interpretation I, the system adds all facts $p_i(a_1, \ldots, a_{m_i}) \leftarrow$ over p_i (with arity m_i) that are true in I to the specified program, creates another instance of the reasoner to evaluate it, and returns a symbolic handle H as result. For convenience, we do not write n in $\&callhex_n$ and $\&callhexfile_n$ as it is understood from the usage.

Example 2. In the following program, we use two predicates p_1 and p_2 to define the input to the subprogram `sub.hex` ($n = 2$), i.e., all atoms over these predicates are added to the subprogram prior to evaluation. The call derives a handle H as result.

$$p_1(x, y) \leftarrow \qquad p_2(a) \leftarrow \qquad p_2(b) \leftarrow$$
$$handle(H) \leftarrow \&callhexfile[\mathtt{sub.hex}, p_1, p_2](H)$$

A *handle* is a unique integer representing a certain cache entry. In the implementation, handles are consecutive numbers starting with 0. Hence in the example the unique answer set of the program is $\{handle(0)\}$ (neglecting facts).

[1] http://www.kr.tuwien.ac.at/research/systems/dlvhex/meld.html

Formally, given an interpretation I, $f_{\&callhexfile_n}(I, \mathit{file}, p_1, \ldots, p_n, h) = 1$ iff h is the handle to the result of the program in file *file*, extended by the facts over predicates $p_1, \ldots, p_n$ that are true in I. The formal notion and use of $\&callhex_n$ to call embedded subprograms is analogous to $\&callhexfile_n$.

Example 3. Consider the following program:

$$h_1(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_2(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_3(H) \leftarrow \&callhex[\texttt{a} \leftarrow . \ \texttt{b} \leftarrow .](H)$$

The rules execute the program `sub.hex` and the embedded program $P_e = \{a \leftarrow,\ b \leftarrow\}$. No facts will be added in this example. The single answer set is $\{h_1(0), h_2(0), h_3(1)\}$ resp. $\{h_1(1), h_2(1), h_3(0)\}$ depending on the order in which the subprograms are executed (which is irrelevant). While $h_1(X)$ and $h_2(X)$ will have the same value for X, $h_3(Y)$ will be such that $Y \neq X$. Our implementation realizes that the result of the program in `sub.hex` is referred to twice but executes it only once; P_e is (possibly) different from `sub.hex` and thus evaluated separately.

Now we want to determine how many (and subsequently which) answer sets it has. For this purpose, we define external atom $\&answersets[PH](AH)$ which maps handles PH to call results to sets of respective answer set handles. Formally, for an interpretation I, $f_{\&answersets}(I, h_P, h_A) = 1$ iff h_A is a handle to an answer set of the program with program handle h_P.

Example 4. The program

$$\mathit{ash}(PH, AH) \leftarrow \&callhex[\texttt{a} \vee \texttt{b} \leftarrow .](PH), \&answersets[PH](AH)$$

calls the embedded subprogram $P_e = \{a \vee b \leftarrow .\}$ and retrieves pairs (PH, PA) of handles to its answer sets. $\&callhex$ returns a handle $PH = 0$ to the result of P_e, which is passed to $\&answersets$. This atom returns a *set* of answer set handles (0 and 1, as P_e has two answer sets, viz. $\{a\}$ and $\{b\}$). The overall program has thus the single answer set $\{\mathit{ash}(0,0), \mathit{ash}(0,1)\}$. As for each program the answer set handles start with 0, only a pair of program and answer set handles uniquely identifies an answer set.

We now are ready to solve our example of counting shortest paths from above.

Example 5. Suppose `paths.hex` is the search program and encodes each shortest path in a separate answer set. Consider the following program:

$$\mathit{as}(AH) \leftarrow \&callhexfile[\texttt{paths.hex}](PH), \&answersets[PH](AH)$$
$$\mathit{number}(D) \leftarrow \mathit{as}(C), D = C + 1, \mathrm{not}\ \mathit{as}(D)$$

The second rule computes the first free handle D; the latter coincides with the number of answer sets of `paths.hex` (assuming that some path between the nodes exists).

At this point we still treat answer sets of subprograms as black boxes. We now define an external atom to investigate them. Given an interpretation I, $f_{\&predicates}(I, h_P, h_A, p, a) = 1$ iff p occurs as an a-ary predicate in the answer set identified by h_P and h_A. Intuitively, the external atom maps pairs of program and answer set handles to the predicates names with their associated arities occurring in the accourding answer set.

Example 6. We illustrate the usage of $\&predicates$ with the following program:

$$\mathit{preds}(P, A) \leftarrow \&callhex[\texttt{node(a).}\ \texttt{node(b).}\ \texttt{edge(a,b).}](PH),$$
$$\&answersets[PH](AH), \&predicates[PH, AH](P, A)$$

It extracts all predicates (and their arities) occurring in the answer of the embedded program P_e, which specifies a graph. The single answer set is $\{\mathit{preds}(\mathit{node}, 1), \mathit{preds}(\mathit{edge}, 2)\}$ as the single answer set of P_e has atoms with predicate node (unary) and edge (binary).

The final step to gather all information from the answer of a subprogram is to extract the *literals* and their *parameters* occurring in a certain answer set. This can be done with external atom $\&arguments$, which is best demonstrated with an example.

Example 7. Consider the following program:

$$\begin{aligned} h(PH, AH) \leftarrow\ & \&callhex[\texttt{node(a). node(b). node(c). edge(a, b).edge(c, a).}](PH), \\ & \&answersets[PH](AH) \\ edge(W, V) \leftarrow\ & h(PH, AH), \&arguments[PH, AH, \texttt{edge}](I, 0, V), \\ & \&arguments[PH, AH, \texttt{edge}](I, 1, W) \\ node(V) \leftarrow\ & h(PH, AH), \&arguments[PH, AH, \texttt{node}](I, 0, V) \end{aligned}$$

It extracts the directed graph given by the embedded subprogram P_e and reverses all edges; the single answer set is $\{h(0,0), node(a), node(b), node(c), edge(b,a), edge(a,c)\}$. Indeed, P_e has a single answer set, identified by $PH = 0, AH = 0$; via $\&arguments$ we can access in the second resp. third rule the facts over $edge$ resp. $node$ in it, which are identified by a unique literal id I; the second output term of $\&arguments$ is the argument position, and the third the actual value at this position. If the predicates of a subprogram were unknown, we can determine them using $\&predicates$.

To check the sign of a literal, the external atom $\&arguments[PH, AH, Pred](I, s, Sign)$ supports argument s. When $s = 0$, $\&arguments$ will match the sign of the I-th *positive* literal over predicate $Pred$ into $Sign$, and when $s = 1$ it will match the corresponding classically negated atom.

4 Applications

MELD. The MELD system [10] deals with merging multiple *collections of belief sets*. Roughly, a belief set is a set of classical ground literals. Practical examples of belief sets include explanations in abduction problems, encodings of decision diagrams, and relational data. The merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms.

The automatic evaluation of tree-shaped merging plans is based on nested HEX-programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. Note that for nesting of ASP-programs with arbitrary (finite) depth, XASP [11] is not appropriate.

Aggregate Functions. Nested programs can also emulate aggregate functions [7] (e.g., sum, count, max) where the (user-defined) host program computes the function given the result of a subprogram. This can be generalized to aggregates over *multiple* answer sets of the subprogram; e.g., to answer set counting, or to find the minimum/maximum of some predicate over all answer sets (which may be exploited for global optimization).

Generalized Quantifiers. Nested HEX-programs make the implementation of brave and cautious reasoning for query answering tasks very easy, even if the backend reasoner only supports answer set enumeration. Furthermore, extended and user-defined types of query answers (cf. [5]) are definable in a very user-friendly way, e.g., majority decisions (at least half of the answer sets support a query), or minimum and/or maximum number based decisions (qualified number restrictions).

Preferences. Answer sets as accessible objects can be easily compared wrt. user-defined preference rules, and used for filtering as well as ranking results (cf. [4]): a host program selects appropriate candidates produced by a subprogram, using preference rules. The latter can be elegantly implemented as ordinary integrity constraints (for filtering), or as rules (possibly involving further external calls) to derive a rank. A popular application are online shops, where the past consumer behavior is frequently used to filter or sort search results. Doing the search via an ASP program which delivers the matches in answer sets, a host program can reason about them and act as a filter or ranking algorithm.

5 Conclusion

To overcome limitations of classical ASP regarding subprograms and reasoning about their possible outcomes, we briefly presented *nested* HEX-*programs*, which realize subprogram calls via special external atoms of HEX-programs; besides modularity, a plus for readability and program reusability, they allow for reasoning over *multiple* answer sets (of subprograms). An prototype implementation on top of DLVHEX is available. Related to this is the work on macros in [2], which allow to call macros in logic programs.

The possibility to access answer sets in a host program, in combination with access to other external computations, makes nested HEX-programs a powerful tool for a number of applications. In particular, libraries and user-defined functions can be incorporated into programs easily. As an interesting aspect is that dynamic program assembly (using a suitable string library) and execution are possible, which other approaches to modular ASP programming do not offer. Exploring this remains for future work.

References

1. Analyti, A., Antoniou, G., Damásio, C.V.: Mweb: a principled framework for modular web rule bases and its semantics. ACM Trans. Comput. Logic 12(2), 17:1–17:46 (2011)
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: ICLP'06, pp. 376–390. (2006)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM (2011), to appear
4. Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Comp. Intell. 20(2), 308–334 (2004)
5. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: LPNMR'97, pp. 290–309. (1997)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in ASP. In: IJCAI'05, pp. 90-96. (2005)
7. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Generat. Comput. 9, 365–385 (1991)
9. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. 35, 813–857 (2009)
10. Redl, C., Eiter, T., Krennwallner, T.: Declarative belief set merging using merging plans. In: PADL'11, pp. 99–114. (2011)
11. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. CoRR abs/1012.5123 (2010), to appear in Theory Pract. Logic Program.

Domain-specific Languages in a Finite Domain Constraint Programming System

Markus Triska

Vienna University of Technology, Vienna, Austria,
triska@dbai.tuwien.ac.at,
WWW home page: http://www.logic.at/prolog/

Abstract. In this paper, we present domain-specific languages (DSLs) that we devised for their use in the implementation of a finite domain constraint programming system, available as `library(clpfd)` in SWI-Prolog and YAP-Prolog. These DSLs are used in propagator selection and constraint reification. In these areas, they lead to concise specifications that are easy to read and reason about. At compilation time, these specifications are translated to Prolog code, reducing interpretative run-time overheads. The devised languages can be used in the implementation of other finite domain constraint solvers as well and may contribute to their correctness, conciseness and efficiency.

Keywords: DSL, code generation, little languages

1 Introduction

Domain-specific languages (DSLs) are languages tailored to a specific application domain. DSLs are typically devised with the goal of increased expressiveness and ease of use compared to general-purpose programming languages in their domains of application ([1]). Examples of DSLs include *lex* and *yacc* ([2]) for lexical analysis and parsing, regular expressions for pattern matching, HTML for document mark-up, VHDL for electronic hardware descriptions and many other well-known instances.

DSLs are also known as "*little languages*" ([3]), where "little" primarily refers to the typically limited intended or main practical application scope of the language. For example, PostScript is a "little language" for page descriptions.

CLP(FD), constraint logic programming over finite domains, is a declarative formalism for describing combinatorial problems such as scheduling, planning and allocation tasks ([5]). It is one of the most widely used instances of the general CLP(·) scheme that extends logic programming to reason over specialized domains. Since CLP(FD) is applied in many industrial settings like systems verification, it is natural to ask: How can we implement constraint solvers that are more reliable and more concise (i.e., easier to read and verify) while retaining their efficiency? In the following chapters, we present little languages that we devised towards this purpose. They are already being used in a constraint solver over finite domains, available as `library(clpfd)` in SWI-Prolog and YAP-Prolog, and can be used in other systems as well.

2 Related work

In the context of CLP(FD), *indexicals* ([4]) are a well-known example of a DSL. The main idea of indexicals is to declaratively describe the domains of variables as functions of the domains of related variables. The indexical language consisting of the constraint "`in`" and expressions such as `min(X)..max(X)` also includes specialized constructs that make it applicable to describe a large variety of arithmetic and combinatorial constraints. GNU Prolog ([7]) and SICStus Prolog ([6]) are well-known Prolog systems that use indexicals in the implementation of their finite domain constraint solvers.

The usefulness of deriving large portions of code automatically from shorter descriptions also motivates the use of *variable views*, a DSL to automatically derive *perfect* propagator variants, in the implementation of Gecode ([8]).

Action rules ([9]) and Constraint Handling Rules ([10]) are Turing-complete languages that are very well-suited for implementing constraint propagators and even entire constraint systems (for example, B-Prolog's finite domain solver).

These examples of DSLs are mainly used for the description and generation of constraint *propagation* code in practice. In the following chapters, we contribute to these uses of DSLs in the context of CLP(FD) systems by presenting DSLs that allow you to concisely express selection of propagators and constraint reification with desirable properties.

3 Matching propagators to constraint expressions

To motivate the DSL that we now present, consider the following quote from Neng-Fa Zhou, author of B-Prolog ([11]):

> A closer look reveals the reason [for failing to solve the problems within the time limit]: Almost all of the failed instances contain non-linear (e.g., $X * Y = C$, $abs(X - Y) = C$, and $X \bmod Y = C$) and disjunctive constraints which were not efficiently implemented in the submitted version of the solver.

Consider the specific example of $abs(X-Y) = C$: It is clear that instead of decomposing the constraint into $X - Y = T$, $abs(T) = C$, a specialized combined propagator can be implemented and applied, avoiding auxiliary variables and intermediate propagation steps to improve efficiency. It is then left to detect that such a specialized propagator can actually be applied to a given constraint expression. This is the task of *matching* available propagators to given constraint expressions, or equivalently, mapping constraint expressions to propagators.

Manually selecting fitting propagators for given constraint expressions is quite error-prone, and one has to be careful not to accidentally unify variables that occur in the expression with subexpressions that one wants to check for. To simplify this task, we devised a DSL in the form of a simple committed-choice language. The language is a list of rules of the form $M \rightarrow As$, where M is a matcher and As is a list of actions that are performed when M matches a posted constraint.

More formally, a *matcher* M consists of the term $m\_c(P, C)$. P denotes a *pattern* involving a constraint *relation* like `#=`, `#>` etc. and its arguments, and C is a *condition* (a Prolog goal) that must hold for a rule to apply. The basic building-blocks of a pattern are explained in Table 1. These building-blocks can be nested inside all symbolic expressions like addition, multiplication etc. A rule is applicable if a given constraint is matched by P (meaning it unifies with P taking the conditions induced by P into account), and additionally C is true. A matcher $m\_c(P, \texttt{true})$, can be more compactly written as $m(P)$.

`any(X)`	Matches any subexpression, unifying X with that expression.
`var(X)`	Matches a variable or integer, unifying X with it.
`integer(X)`	Matches an integer, unifying X with it.

Table 1. Basic building-blocks of a pattern

In a rule $M \rightarrow As$, each action A_i in the list of actions $As = [A_1, \ldots, A_n]$ is one of the actions described in Table 2. When a rule is applicable, its actions are performed in the order they occur in the list, and no further rules are tried.

Figure 1 shows some of the matching rules that we use in our constraint system. It is only an excerpt; for example, in the actual system, nested additions are also detected and handled by a dedicated propagator. Such a declarative description has several advantages: First, it allows automated subsumption checks to detect whether specialized propagators are accidentally overshadowed by other rules. This is also a mistake that we found easy to make and hard to detect when manually selecting propagators. Second, when DSLs similar to the one we propose here are also used in other constraint systems, it is easier to compare supported specialized propagators, and to support common ones more uniformly across systems. Third, improvements to the expansion phase of the DSL benefits potentially many propagators at once.

`g(G)`	Call the Prolog goal G.
`d(X, Y)`	Decompose arithmetic subexpression X, unifying Y with its result. Equivalent to `g(parse_clpfd(X, Y))`, an internal predicate that is also generated from a similar DSL.
`p(P)`	Post a constraint propagator P. This is a shorthand notation for a specific sequence of goals that add a constraint to the constraint store and trigger it.
`r(X, Y)`	Rematch the rule's constraint relation, using arguments X and Y. Equivalent to `g(call(F,X,Y))`, where F is the functor of the rule's pattern.

Table 2. Valid actions in a list As of a rule $M \rightarrow As$

```
1  m(integer(I) #>= abs(any(X))) => [d(X, RX), g((I>=0, I1 is -I, RX in I1..I))]
2  m(any(X) #>= any(Y))          => [d(X, RX), d(Y, RY), g(geq(RX, RY))]
3
4  m(var(X) #= var(Y)+var(Z))    => [p(pplus(Y,Z,X))]
5  m(var(X) #= var(Y)-var(Z))    => [p(pplus(X,Z,Y))]
6  m(any(X) #= any(Y))           => [d(X, RX), d(Y, RX)]
7
8  m(var(X) #\= integer(Y))      => [g(neq_num(X, Y))]
9  m(any(X) #\= any(Y) + any(Z)) => [d(X, X1), d(Y, Y1), d(Z, Z1),
10                                    p(x_neq_y_plus_z(X1, Y1, Z1))]
11 m(any(X) #\= any(Y) - any(Z)) => [d(X, X1), d(Y, Y1), d(Z, Z1),
12                                    p(x_neq_y_plus_z(Y1, X1, Z1))]
13 m(any(X) #\= any(Y))          => [d(X, RX), d(Y, RY), g(neq(RX, RY))]
```

Fig. 1. Rules for matching propagators in our constraint system. (Excerpt)

We found that the languages features we introduced above for matchers and actions enable matching a large variety of intended specialized propagators in practice, and believe that other constraint systems may benefit from this or similar syntax as well.

4 Constraint reification

We now present a DSL that simplifies the implementation of constraint *reification*, which means reflecting the truth values of constraint relations into Boolean $0/1$-variables.

When implementing constraint reification, it is tempting to proceed as follows: For concreteness, consider reified equality (`#=/2`) of two CLP(FD) expressions A and B. We could introduce two temporary variables, T_A and T_B, and post the constraints T_A `#= A` and T_B `#= B`, thus using the constraint solver itself to decompose the (possibly compound) expressions A and B, and reducing reified equality of two *expressions* to equality of two finite domain *variables* (or integers), which is easier to implement. Unfortunately, this strategy yields wrong results in general. Consider for example the constraint (`#<==>/2` denotes Boolean equivalence):

```
(X/0 #= Y/0) #<==> B
```

It is clear that the relation `X/0 #= Y/0` cannot hold, since a divisor can never be 0. A valid (declaratively equivalent) answer to the above constraint is thus (note that X and Y must be constrained to integers for the relation to hold):

```
B = 0, X in inf..sup, Y in inf..sup
```

However, if we decompose the equality `X/0 #= Y/0` into two auxiliary constraints T_A `#= X/0` and T_B `#= Y/0` and post them, then (with strong enough propagation of division) both auxiliary constraints fail, and thus the whole query (incorrectly) fails. While devising a DSL for reification, we found one commercial Prolog system and one freely available system that indeed incorrectly failed in this case. After we reported the issue, the problem was immediately fixed.

It is thus necessary to take *definedness* into account when reifying constraints. See also [12], where our constraint system (in contrast to others that were tested) correctly handles all reification test cases, which we attribute in part to the DSL presented in this chapter. Once any subexpression of a relation becomes undefined, the relation cannot hold and its associated truth value must be 0. Undefinedness can occur when $Y = 0$ in the expressions X/Y, $X \bmod Y$, and $X \operatorname{rem} Y$. Parsing an arithmetic expression that

occurs as an argument of a constraint that is being reified is thus at least a ternary relation, involving the expression itself, its arithmetic result, and its Boolean definedness.

There is a fourth desirable component in addition to those just mentioned: It is useful to keep track of *auxiliary variables* that are introduced when decomposing subexpressions of a constraint that is being reified. The reason for this is that the truth value of a reified constraint may turn out to be irrelevant, for instance the implication `0 #==>` C holds for both possible truth values of the constraint C, thus auxiliary variables that were introduced to hold the results of subexpressions while parsing C can be eliminated. However, we need to be careful: A constraint propagator may *alias* user-specified variables with auxiliary variables. For example, in `abs(X) #= T, X #>= 0`, a constraint system may deduce `X = T`. Thus, if T was previously introduced as an auxiliary variable, and X was user-specified, X must still retain its status as a constrained variable.

These considerations motivate the following DSL for parsing arithmetic expressions in reified constraints, which we believe can be useful in other constraint systems as well: A parsing rule is of the form $H \rightarrow Bs$. A head H is either a term $g(G)$, meaning that the Prolog goal G is true, or a term $m(P)$, where P is a symbolic pattern and means that the expression E that is to be parsed can be decomposed as stated, recursively using the parsing rules themselves for subterms of E that are subsumed by variables in P. The body Bs of a parsing rule is a list of body elements, which are described in Table 3. The predicate `parse_reified/4`, shown in Figure 2, contains our full declarative specification for parsing arithmetic expressions in reified constraints, relating an arithmetic expression E to its result R, Boolean definedness D, and auxiliary variables according to the given parsing rules, which are applied in the order specified, committing to the first rule whose head matches. This specification is again translated to Prolog code at compile time and used in other predicates.

`g(G)`	Call the Prolog goal G.
`d(D)`	D is 1 if and only if all subexpressions of E are defined.
`p(P)`	Add the constraint propagator P to the constraint store.
`a(A)`	A is an auxiliary variable that was introduced while parsing the given compound expression E.
`a(X,A)`	A is an auxiliary variable, unless `A == X`.
`a(X,Y,A)`	A is an auxiliary variable, unless `A == X` or `A == Y`.
`skeleton(Y,D,G)`	A "skeleton" propagator is posted. When Y cannot become 0 any more, it calls the Prolog goal G and binds $D = 1$. When Y is 0, it binds $D = 0$. When $D = 1$ (i.e., the constraint must hold), it posts `Y #\= 0`.

Table 3. Valid body elements for a parsing rule

```
parse_reified(E, R, D,
    [g(cyclic_term(E)) => [g(domain_error(clpfd_expression, E))],
     g(var(E))         => [g((constrain_to_integer(E), R=E, D=1))],
     g(integer(E))     => [g((R=E, D=1))],
     m(-X)             => [d(D), p(ptimes(-1,X,R)), a(R)],
     m(abs(X))         => [g(R#>=0), d(D), p(pabs(X, R)), a(X,R)],
     m(X+Y)            => [d(D), p(pplus(X,Y,R)), a(X,Y,R)],
     m(X-Y)            => [d(D), p(pplus(R,Y,X)), a(X,Y,R)],
     m(X*Y)            => [d(D), p(ptimes(X,Y,R)), a(X,Y,R)],
     m(X^Y)            => [d(D), p(pexp(X,Y,R)), a(X,Y,R)],
     m(min(X,Y))       => [d(D), p(pgeq(X, R)), p(pgeq(Y, R)),
                           p(pmin(X,Y,R)), a(X,Y,R)],
     m(max(X,Y))       => [d(D), p(pgeq(R, X)), p(pgeq(R, Y)),
                           p(pmax(X,Y,R)), a(X,Y,R)],
     m(X/Y)            => [skeleton(Y,D,X/Y #= R)],
     m(X mod Y)        => [skeleton(Y,D,X mod Y #= R)],
     m(X rem Y)        => [skeleton(Y,D,X rem Y #= R)],
     g(true)           => [g(domain_error(clpfd_expression, E))]]).
```

Fig. 2. Parsing arithmetic expressions in reified constraints with our DSL

5 Conclusion and future work

We have presented DSLs that are used in the implementation of a finite domain constraint programming system. They enable us to capture the intended functionality with concise declarative specifications. We believe that identical or similar DSLs are also useful in the implementation of other constraint systems. In the future, we intend to generate even more currently hand-written code automatically from smaller declarative descriptions.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Comput. Surv., 37(4), 316–344 (2005)
2. Johnson, S. C., Lesk, M. E.: Language development tools. Bell System Technical Journal, 56(6), 2155–2176 (1987)
3. Bentley, J.: Little languages. Communications of the ACM, 29(8), 711–21 (1986)
4. Codognet, P., Diaz, D.: Compiling Constraints in clp(FD). Journal of Logic Programming, 27(3) (1996)
5. Jaffar, J., Lassez, J-L.: Constraint Logic Programming. POPL, 111–119 (1987)
6. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. Proc. Prog. Lang.: Implementations, Logics, and Programs (1997)
7. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming (JFLP), Vol. 2001, No. 6 (2001)
8. Schulte, Ch., Tack, G.: Perfect Derived Propagators. CoRR entry (2008)
9. Zhou, N-F.: Programming Finite-Domain Constraint Propagators in Action Rules. Theory and Practice of Logic Programming, Vol.6, No.5, pp. 483–508 (2006)
10. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Special Issue on Constraint Logic Programming, J. of Logic Programming, Vol 37(1–3) (1998)
11. Zhou, N-F.: A Report on the BPrologCSP Solver (2007)
12. Frisch, Alan M., Stuckey, Peter J.: The Proper Treatment of Undefinedness in Constraint Languages, CP 2009, LNCS 5732, 367–382 (2009)

Computing with Logic as Operator Elimination: The ToyElim System

Christoph Wernhard

Technische Universität Dresden
christoph.wernhard@tu-dresden.de

Abstract. A prototype system is described whose core functionality is, based on propositional logic, the elimination of second-order operators, such as Boolean quantifiers and operators for projection, forgetting and circumscription. This approach allows to express many representational and computational tasks in knowledge representation – for example computation of abductive explanations and models with respect to logic programming semantics – in a uniform operational system, backed by a uniform classical semantic framework.

1 Computation with Logic as Operator Elimination

We pursue an approach to computation with logic emerging from three theses:

> *1. Classical first-order logic extended by some second-order operators suffices to express many techniques of knowledge representation.*

Like the standard logic operators, second-order operators can be defined *semantically*, by specifying the requirements on an interpretation to be a model of a formula whose principal functor is the operator, depending only on semantic properties of the argument formulas. Neither control structure imposed over formulas (e.g. Prolog), nor formula transformations depending on a particular syntactic shape (e.g. Clark's completion) are involved. Compared to classical first-order formulas, the second-order operators give additional expressive power. Circumscription is a prominent knowledge representation technique that can be expressed with second-order operators, in particular predicate quantifiers [1].

> *2. Many computational tasks can be expressed as elimination of second-order operators.*

Elimination is a way to computationally process second-order operators, for example Boolean quantifiers with respect to propositional logic: The input is a formula which may contain the operator, for example a quantified Boolean formula such as $\exists q\ ((p \leftarrow q) \wedge (q \leftarrow r))$. The output is a formula that is equivalent to the input, but in which the operator does not occur, such as, with respect to the formula above, the propositional formula $p \leftarrow r$. Let us assume that the method used to eliminate the Boolean quantifiers returns formulas in which not just the quantifiers but also the quantified propositional variables do not occur. This syntactic condition is usually met by elimination procedures. Our method then subsumes a variety of tasks: Computation of uniform interpolants, QBF and SAT solving, as well as computation of certain forms of abductive explanations, of propositional circumscription, and of stable models, as will be outlined below.

> *3. Depending on the application, outputs of computation with logic are conveniently represented by formulas meeting syntactic criteria.*

If results of elimination are formulas characterized just up to semantics, they may contain redundancies and be in a shape that is difficult to comprehend. Thus, they should be subjected to simplification and canonization procedures before passed to humans or machine clients. The output format depends on the application problem: What is a CNF of the formula? Are certain facts consequences of the formula? What are the models of the formula? What are its minimal models? What are its 3-valued models with respect to some encoding into 2-valued logics? Corresponding answers can be computed on the basis of normal form representations of the elimination outputs: CNFs, DNFs, and full DNFs. Of course, transformation into such normal forms might by itself be an expensive task. Second-order operators allow to counter this by specifying a small set of application relevant symbols that should be included in the output (e.g. by Boolean quantification upon the irrelevant atoms).

2 Features of the System

ToyElim[1] is a prototype system developed to investigate operator elimination from a pragmatic point of view with small applications. For simplicity, it is based on propositional logic, although its characteristic features should transfer to first-order logic. It supports a set of second-order operators that have been semantically defined in [11, 14, 13].

Formula Syntax. As the system is implemented in Prolog, formulas are represented by Prolog terms, the standard connectives corresponding to `true`/0, `false`/0, `~`/1, `,`/2, `;`/2, `->`/2, `<-`/2, `<->`/2. Propositional atoms are represented by Prolog atoms or compound ground terms. The system supports propositional expansion with respect to finite domains of formulas containing first-order quantifiers.

Forgetting. Existential Boolean quantification $\exists p\ F$ can be expressed as *forgetting* [11, 4] in formula F about atom p, written $\mathsf{forget}_{\{p\}}(F)$, represented by `forg([p], F')` in system syntax, where F' is the system representation of F. To get an intuition of forgetting, consider the equivalence $\mathsf{forget}_{\{p\}}(F) \equiv F[p\backslash\mathsf{true}] \vee F[p\backslash\mathsf{false}]$, where $F[p\backslash\mathsf{true}]$ ($F[p\backslash\mathsf{false}]$) denotes F with all occurrences of p replaced by true (false). Rewriting with this equivalence constitutes a naive method for eliminating the forgetting operator. The formula $\mathsf{forget}_{\{p\}}(F)$ can be said to express the same as F about all other atoms than p, but nothing about p.

Elimination and Pretty Printing of Formulas. The central operation of the ToyElim system, elimination of second-order operators, is performed by the predicate `elim(F,G)`, with input formula F and output formula G. For example, define as extension of `kb1`/1 a formula (after [3]) as follows:

```
kb1(((shoes_are_wet <- grass_is_wet),                    (1)
     (grass_is_wet <- rained_last_night),
     (grass_is_wet <- sprinkler_was_on))).
```

After consulting this, we can execute the following query on the Prolog toplevel:

```
?- kb1(F), elim(forg([grass_is_wet], F), G), ppr(G).    (2)
```

This results in binding `G` to the output of eliminating the forgetting about `grass_is_wet`. The predicate `ppr`/1 is one of several provided predicates for converting formulas into application adequate shapes. It prints its argument as CNF with clauses written as reverse implications:

```
((shoes_are_wet <- rained_last_night),                   (3)
 (shoes_are_wet <- sprinkler_was_on)).
```

Scopes. So far, the first argument of forgetting has been a singleton set. More generally, it can be an arbitrary set of atoms, corresponding to nested existential quantification. Even more generally, also polarity can be considered: Forgetting can, for example, be applied only to those occurrences of an atom which have negative polarity in a NNF formula. This can be expressed by *literals* with explicitly written sign in the first argument of the forgetting operator. Forgetting about an atom is equivalent to nested forgetting about the positive and the negative literal with that atom. In accord with this observation, we technically consider the first argument of forgetting always as a *set of literals*, and regard an unsigned atom there as a shorthand representing both of its literals. For example, `[+grass_is_wet, shoes_are_wet]` is a shorthand for `[+grass_is_wet, +shoes_are_wet, -shoes_are_wet]`. Not just forgetting, but, as shown below, also other second-order operators have a set of literals as parameter. Hence, we refer to a set of literals in this context by a special name, as *scope*.

Projection. In many applications it is useful to make explicit not the scope that is "forgotten" about, but what is preserved. The *projection* [11] of formula F onto scope S, which can be defined for scopes S and formulas F as $\mathsf{project}_S(F) \equiv \mathsf{forget}_{\mathsf{ALL}-S}(F)$, where ALL denotes the set of all literals, serves this purpose. Vice versa, forgetting could be defined in terms of projection: $\mathsf{forget}_S(F) \equiv \mathsf{project}_{\mathsf{ALL}-S}(F)$. The call to `elim`/2 in the query (2) can equivalently be expressed with projection instead of forgetting by

```
elim(proj([shoes_are_wet, rained_last_night, sprinkler_was_on], F).   (4)
```

[1] http://cs.christophwernhard.com/provers/toyelim/, under GNU Public License.

User Defined Logic Operators – An Example of Abduction. ToyElim allows the user to specify macros for use in the input formulas of `elim`/2. The following example extends the system by a logic operator `gwsc` for a variant of the weakest necessary condition [8], characterized in terms of projection:

```
:- define_elim_macro(gwsc(S, F, G), ~proj(complements(S), (F, ~G))).        (5)
```

Here `complements(S)` specifies the set of the literal complements of the members of the scope specified by `S`. The term `gwsc(S, F, G)` is the system syntax for $\mathsf{gwsc}_S(F, G)$, the *globally weakest sufficient condition* of formula G on scope S within formula F, which satisfies the following: A formula H is equivalent to $\mathsf{gwsc}_S(F, G)$ if and only if it holds that (1.) $H \equiv \mathsf{project}_S(H)$; (2.) $F \models H \rightarrow G$; (3.) For all formulas H' such that $H' \equiv \mathsf{project}_S(H')$ and $F \models G \rightarrow H'$ it holds that $H \models H'$. With the `gwsc` operator certain abductive tasks [3] can be expressed. The following query, for example, yields abductive explanations for `shoes_are_wet` in terms of {`rained_last_night`, `sprinkler_was_on`} with respect to the knowledge base (1):

```
?- kb1(F),                                                                  (6)
   elim(gwsc([rained_last_night, sprinkler_was_on], F, shoes_are_wet),
        G),
   ppm(G).
```

The predicate `ppm`/1 serves, like `ppr`/1, to convert formulas to application adequate shape. It writes a DNF of its input, in list notation, and simplified such that it does not contain tautologies and subsumed clauses. In the example the output has two clauses, each representing an alternate explanation:

```
[[rained_last_night],[sprinkler_was_on]].                                   (7)
```

Scope-Determined Circumscription. A further second-order operator supported by ToyElim is *scope-determined circumscription* [14]. The corresponding functor `circ` has, like `proj` and `forg`, a scope specifier and a formula as arguments. It allows to express *parallel predicate circumscription with varied predicates* [5] (only propositional, since the system is based on propositional logic). The scope specifier controls the effect of circumscription: Atoms that occur just in a *positive* literal in the scope are minimized; symmetrically, atoms that occur just *negatively* are maximized; atoms that occur in *both polarities* are fixed; and atoms that do *not at all* occur in the scope are allowed to vary. For example, the scope specifier, `[+abnormal, bird]`, a shorthand for `[+abnormal, +bird, -bird]`, expresses that `abnormal` is minimized, `bird` is fixed, and all other predicates are varied.

Predicate Groups and Systematic Renaming. Semantics for knowledge representation sometimes involve what might be described as handling different occurrences of a predicate differently – for example depending on whether it is subject to negation as failure. If such semantics are to be modeled with classical logic, then these occurrences can be identified by using distinguished predicates, which are equated with the original ones when required. To this end, ToyElim supports the handling of *predicate groups*: The idea is that each predicate actually is represented by several *corresponding* predicates $p^0, \ldots, p^n$, where the superscripted index is called *predicate group*. In the system syntax, the predicate group of an atom is represented within its main functor: If the group is larger than 0, the main functor is suffixed by the group number; if it is 0, the main functor does not end in a number. For example $p(a)^0$ and $p(a)^1$ are represented by `p(a)` and `p1(a)`, respectively. In scope specifiers, a number is used as shorthand for the set of all literals whose atom is from the indicated group, and a number in a sign functor for the set of those literals which have that sign and whose atom is from the indicated group. For example, `[+(0), 1]` denotes the union of the set of all positive literals whose atom is from group 0 and of the set of all literals whose atom is from group 1. Systematic renaming of all atoms in a formula that have a specific group to their correspondents from another group can be expressed in terms of forgetting [13]. The ToyElim system provides the second-order operator `rename` for this. For example, `rename([1-0], F)` is equivalent to F after eliminating second-order operators, followed by replacing all atoms from group 1 with their correspondents from group 0.

An Example of Modeling a Logic Programming Semantics. Scope-determined circumscription and predicate groups can be used to express the characterization of the stable models semantics in terms of circumscription [7] (described also in [6, 13]). Consider the following knowledge base:

```
kb2(((shoes_are_wet <- grass_is_wet),                                    (8)
     (grass_is_wet <- sprinkler_was_on, ~sprinkler_was_abnormal1),
     sprinkler_was_on)).
```

Group 1 is used here to indicate atoms that are subject to negation as failure: All atoms in (8) are from group 0, except for `sprinkler_was_abnormal1`, which is from 1. The user defined operator `stable` renders the stable models semantics:

```
:- define_elim_macro(stable(F), rename([1-0], circ([+(0),1], F))).       (9)
```

The following query then yields the stable models:

```
:- kb2(F), elim(stable((F)), G), ppm(G).                                 (10)
```

The result is displayed with `ppm`/1, as in query (6). It shows here a DNF with a single clause, representing a single model. The positive members of the clause constitute the answer set

```
[[grass_is_wet, shoes_are_wet, ~sprinkler_was_abnormal, sprinkler_was_on]]. (11)
```

If it is only of interest whether `shoes_are_wet` is a consequence of the knowledge base under stable models semantics, projection can be applied to obtain a smaller result. The query

```
:- kb2(F), elim(proj([shoes_are_wet], stable(F)), G), ppm(G).            (12)
```

will effect that the DNF `[[shoes_are_wet]]` is printed.

3 Implementation

The ToyElim system is implemented in SWI-Prolog and can invoke external systems such as SAT and QBF solvers. It runs embedded in the Prolog environment, allowing for example to pass intermediate results between its components through Prolog variables, as exemplified by the queries shown above.

The implementation of the core predicate `elim`/2 maintains a formula which is gradually rewritten until it contains no more second-order operators. It is initialized with the input formula, preprocessed such that only two primitively supported second-order operators remain: forgetting and renaming. It then proceeds in a loop where alternately equivalence preserving simplifying rewritings are applied, and a subformula is picked and handed over for elimination to a specialized procedure. The simplifying rewritings include distribution of forgetting over subformulas and elimination steps that can be performed with low cost [12]. Rewriting of subformulas with the Shannon expansion enables low-cost elimination steps. It is performed at this stage if the expansion, combined with low-cost elimination steps and simplifications, does not lead to an increase of the formula size. The subformula for handing over to a specialized method is picked with the following priority: First, an application of forgetting upon the whole signature of a propositional argument, which can be reduced by a SAT solver to either true or false, is searched. Second, a subformula that can be reduced analogously by a QBF solver, and finally a subformula which properly requires elimination of forgetting. For the latter, ToyElim schedules a portfolio of different methods, where currently two algorithmic approaches are supported: Resolvent generation (SCAN, Davis-Putnam method) and rewriting of subformulas with the Shannon expansion [10, 12]. Recent SAT preprocessors partially perform variable elimination by resolvent generation. *Coprocessor* [9] is such a preprocessor that is configurable such that it can be invoked by ToyElim for the purpose of performing the elimination of forgetting.

4 Conclusion

We have seen a prototype system for computation with logic as elimination of second-order operators. The system helped to concretize requirements on the user interface and on processing methods of systems which are entailed by that approach. In the long run, such a system should be based on more expressive logics than propositional logic. ToyElim is just a first pragmatic attempt, taking advantage of recent advances in SAT solving. A major difference in a first-order setting is that computations of elimination tasks then inherently do not terminate for all inputs.

A general system should for special subtasks not behave worse than systems specialized for these. This can be achieved by identifying such subtasks, or by general methods that implicitly operate like the specialized ones. ToyElim identifies SAT and QBF subtasks. It is a challenge to extend this range, for example, such that the encoded stable model computation would be performed efficiently. The system picks in each round a single subtask that is passed to a specialized solver. We plan to experiment with a more flexible regime, where different subtasks are alternately tried with increasing timeouts.

Research on the improvement of elimination methods includes further consideration of techniques from SAT preprocessors, investigation of tableau and DPLL-like techniques [12, 2], and, in the context of first-order logic, the so called *direct methods* [1]. In addition, it seems worth to investigate further types of output: incremental construction, like enumeration of model representations, and representations of proofs.

The approach of computation with logic by elimination leads to a system that provides a uniform user interface covering many tasks, like satisfiability checking, computation of abductive explanations and computation of models for various logic programming semantics. Variants of established concepts can be easily expressed on a clean semantic basis and made operational. The approach supports the co-existence of different knowledge representation techniques in a single system, backed by a single classical semantic framework. This seems a necessary precondition for logic libraries that accumulate knowledge independently of some particular application.

References

1. D. M. Gabbay, R. A. Schmidt, and A. Szałas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, 2008.
2. J. Huang and A. Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI-05*, pages 156–162. Morgan Kaufmann, 2005.
3. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
4. J. Lang, P. Liberatore, and P. Marquis. Propositional independence – formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
5. V. Lifschitz. Circumscription. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.
6. V. Lifschitz. Twelve definitions of a stable model. In *ICLP 2008*, volume 5366 of *LNCS*, pages 37–51. Springer, 2008.
7. F. Lin. *A Study of Nonmonotonic Reasoning*. PhD thesis, Stanford University, 1991.
8. F. Lin. On strongest necessary and weakest sufficient conditions. *Artificial Intelligence*, 128:143–159, 2001.
9. N. Manthey. Coprocessor - a standalone SAT preprocessor. In *WLP 2011*, 2011. (This volume).
10. N. V. Murray and E. Rosenthal. Tableaux, path dissolution and decomposable negation normal form for knowledge compilation. In *TABLEAUX 2003*, volume 2796 of *LNAI*, pages 165–180. Springer, 2003.
11. C. Wernhard. Literal projection for first-order logic. In *JELIA 08*, volume 5293 of *LNAI*, pages 389–402. Springer, 2008.
12. C. Wernhard. Tableaux for projection computation and knowledge compilation. In *TABLEAUX 2009*, volume 5607 of *LNAI*, pages 325–340. Springer, 2009.
13. C. Wernhard. Circumscription and projection as primitives of logic programming. In *Tech. Comm. of the ICLP 2010*, volume 7 of *LIPIcs*, pages 202–211, 2010.
14. C. Wernhard. Literal projection and circumscription. In *FTP'09*, volume 556 of *CEUR Workshop Proceedings*, pages 60–74. CEUR-WS.org, 2010.

Coprocessor - a Standalone SAT Preprocessor

Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

Abstract. In this work a stand-alone preprocessor for SAT is presented that is able to perform most of the known preprocessing techniques. Preprocessing a formula in SAT is important for performance since redundancy can be removed. The preprocessor is part of the SAT solver *riss* [9] and is called *Coprocessor*. Not only *riss*, but also *MiniSat 2.2* [11] benefit from it, because the *SatELite* preprocessor of *MiniSat* does not implement recent techniques. By using more advanced techniques, *Coprocessor* is able to reduce the redundancy in a formula further and improves the overall solving performance.

1 Introduction

In theory SAT problems with n variables have a worst case execution time of $O(2^n)$ [2]. Reducing the number of variables results in a theoretically faster search. However, in practice the number of variables does not correlate with the runtime. The number of clauses highly influences the performance of unit propagation. Preprocessing helps to reduce the size of the formula by removing variables and clauses that are redundant. Due to limited space it is assumed that the reader is familiar with basic preprocessing techniques [3]. Preprocessing techniques can be classified into two categories: Techniques, which change a formula in a way that the satisfying assignment for the preprocessed formula is not necessarily a model for the original formula, are called *satisfiability-preserving techniques*. Thus, for these techniques undo information has to be stored. For the second category, this information is not required. The second category is called *equivalence-preserving techniques*, because the preprocessed and original formula are equivalent.

This paper is structured in the following way. An overview of the implemented techniques is given in Sect. 2. Details on *Coprocessor*, a format for storing the undo information and a comparison to *SatELite* is given in Sect. 3. Finally, a conclusion is given in Sect. 4.

2 Preprocessor Techniques

The notation used to describe the preprocessor is the following: variables are numbers and literals are positive or negative variables, e.g. 2 and $\neg 2$. A clause C is a disjunction of a set of literals, denoted by $[l_1, \ldots, l_n]$. A formula is a conjunction of clauses. The original formula will be referred to as F, the preprocessed formula is always called F'. Unit propagation on F is denoted by BCP(l), where l is the literal that is assigned to *true*.

2.1 Satisfiability-Preserving Techniques

The following techniques change F in a way, that models of F' are no model for F anymore. Therefore, these methods need to store undo information. Undoing of these methods has to be done carefully, because the order influences the resulting assignment. All the elimination steps have to be undone in the opposite order they have been applied before [6].

Variable Elimination (VE) [3,13] is a technique to remove variables from the formula. Removing a variable is done by resolving the according clauses in which the variable occurs. Given two sets of clauses: C_x with the positive variable x and $C_{\overline{x}}$ with negative x. Let G be the union of these two sets $G \equiv C_x \cup C_{\overline{x}}$. Resolving these two sets on variable x results in a new set of clauses G' where tautologies are not included. It is shown in [3] that G can be replaced by G' without changing the satisfiability of the formula. If a model

is needed for the original formula, then the partial model can be extended using the original clauses F to assign variable x. Usually, applying VE to a variable results in a larger number of clauses. In state-of-the-art preprocessors VE is only applied to a variable if the number of clauses does not increase. The resulting formula depends on the order of the eliminated variables. *Pure literal elimination* is a special case of VE, because the number of resolvents is zero.

Blocked Clause Elimination (BCE) [7] removes redundant *blocked clauses*. A clause C is blocked if it contains a blocking literal l. A literal l is a blocking literal, if $\bar{l}$ is part of C, and for each clause $C' \in F$ with $\bar{l} \in C'$ the resolvent $C \otimes_l C'$ is a tautology [4,7]. Removing a blocked clause from F changes the satisfying assignments [4]. Since BCE is confluent, the order of the removals does not change the result [7].

Equivalence Elimination (EE) [5] removes a literal l if it is equivalent to another literal l'. Only one literal per equivalence class is kept. Equivalent literals can be found by finding strongly connected components in the binary implication graph (BIG). The BIG represents all implications in the formula by directed edges $l \rightarrow l'$ between literals that occur in a clause [$\bar{l}, l'$]. If a cycle $a \rightarrow b \rightarrow c \rightarrow a$ is found, there is also a cycle $\overline{a} \rightarrow \overline{b} \rightarrow \overline{c} \rightarrow \overline{a}$ and therefore $a \equiv b \equiv c$ can be shown and applied to F by replacing b, and c by a. Finally, double literal occurrences and tautologies are removed.

Let F be $\langle [1, \neg 2]_1, [\neg 1, 2]_2, [1, 2, 3]_3, [\neg 1, \neg 3]_4, [\neg 3, 4]_5, [\neg 1, \neg 4]_6 \rangle$. The index i of a clause C_i gives the position of the clause in the formula. The order to apply techniques is EE, VE and finally BCE. EE will find $1 \equiv 2$ based on the clauses C_1 and C_2. Thus, it replaces each occurrence of 2 with 1, since 1 is the smaller variable. This step alters C_3 to $C_7 = [1, 3]$. Now VE on variable 3 detects that there are 3 clauses in which 3 occurs. The single resolvent that can be build is $C_{7 \otimes 5} = [1, 4]$. Finally, BCE removes the two clauses, because all literals of each clause are blocking literals. Since the resulting formula is empty, it is satisfied by any interpretation. It can be clearly seen, that the original formula cannot be satisfied by any interpretation.

2.2 Equivalence-Preserving Techniques

Equivalence-preserving techniques can be applied in any order, because the preprocessed formula is equivalent to the original one. By combining the following techniques with satisfiability-preserving techniques the order of the applied techniques has to be stored, to be able to undo all changes correctly.

Hidden Tautology Elimination (HTE) [4] is based on the clause extension *hidden literal addition* (HLA). After the clause C is extended by HLA, C is removed if it is tautology. The HLA of a clause C with respect to a formula F is computed as follows: Let l be a literal of C and $[l', l] \in F \setminus \{C\}$. If such a literal l' can be found, C is extended by $C := C \cup \overline{l'}$. This extension is applied until fix point. HLA can be regarded as the opposite operation of self subsuming resolution. The algorithm is linear time in the number of variables [4]. An example for HTE is the formula $F = \langle [1, 3], [-2, 3], [1, 2] \rangle$. Extending the clause C_1 stepwise can look as follows: $C_1 = [1, 3, \neg 2]$ with C_3. Next, $C_1 = [1, 3, \neg 2, 2]$ with C_2, so that it becomes tautology and can be removed.

Probing [8] is a technique to simplify the formula by propagating variables in both polarities l and $\bar{l}$ separately and comparing their implications or by propagating all literals of a clause $C = [l_1, \ldots, l_n]$, because it is known that in the two cases one of the candidates has to be satisfied.

Probing a single variable can find a conflict and thus finds a new unit. The following example illustrates the other cases:

$$\text{BCP}(1) \Rightarrow 2, 3, 4, \neg 5, \neg 7$$
$$\text{BCP}(\overline{1}) \Rightarrow 2, \neg 4, 6, 7$$

To create a complete assignment, variable *1* has to be assigned and both possible assignments imply 2, so that 2 can be set to *true* immediately. Furthermore, the equivalences $4 \equiv 1$ and $\overline{7} \equiv 1$ can be found. These equivalences can also be eliminated. Probing all literals of a clause can find only new units.

Vivification (also called *Asymmetric Branching*) [12] reduces the length of a clause by propagating the negated literals of a clause $C = [l_1, \ldots, l_n]$ iteratively until one of the following three cases occurs:

1. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) results in an empty clause for $i < n$.
2. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) implies another literal l_j of the C with $i < j < n$
3. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) implies another negated literal $\overline{l_j}$ of the C with $i < j \leq n$

In the first case, the unsatisfying partial assignment is disallowed by adding a clause $C' = [l_1, \ldots, l_i]$. The clause C' subsumes C. The implication $\overline{l_1} \wedge \cdots \wedge \overline{l_i} \rightarrow l_j$ in the second case results in the clause $C' = [l_1, \ldots, l_i, l_j]$ that also subsumes C. Formulating the third case into a clause $C' = [l_1, \ldots, l_i, \overline{l_j}]$ subsumes C by applying self subsumption to $C'' = C \otimes_{l_j} C' = [l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_n]$.

Extended Resolution (ER) [1] introduces a new variables v to a formula that is equivalent to a disjunction of literals $v \equiv l \vee l'$. All clauses in F are updated by removing the pair and adding the new variable instead. It has been shown, that ER is good for shrinking the proof size for unsatisfiable formulas. Applying ER during search as in [1] resulted in a lower performance of *riss*, so that this technique has been put into the preprocessor and replaces the most frequent literal pairs. Still, no deep performance analysis has been done on this technique in the preprocessor, but it seems to boost the performance on unsatisfiable instances.

3 Coprocessor

The preprocessor of *riss*, *Coprocessor*, implements all the techniques presented in Sect. 2 and introduces many algorithm parameters. A description of these parameters can be found in the help of *Coprocessor*[1]. The techniques are executed in a loop on F, so that for example the result of HTE can be processed with VE and afterwards HTE tries to eliminate clauses again.

It is possible to maintain a blacklist and a white-list of variables. Variables on the white-list are tabooed for any non-model-preserving techniques so that their semantic is the same in F'. Variables on the blacklist are always removed by VE.

Furthermore, the resulting formula can be compressed. If variables are removed or are already assigned a value, the variables of the reduct of F' are usually not dense any more. Giving the reduct to another solver increases its memory usage unnecessarily. To overcome this weakness, a compressor has been built into *Coprocessor* that fills these gaps with variables that still occur in F' and stores the already assigned variables for postprocessing the model. The compression cannot be combined with the white-list.

Another transformation that can be applied by the presented preprocessor is the conversion from encoded CSP domains from the direct encoding to the regular encoding as described in [10].

3.1 The Map File Format

A map file is used to store the preprocessing information that is necessary to postprocess a model of F' such that it becomes a model for F again. The map file and the model for F' can be used to restore the model for F by giving this information to *Coprocessor*. The following information has to be stored to be able to do so:

Once	Per elimination step
Compression Table	Variable Elimination
Equivalence Classes	Blocked Clause Elimination
	Equivalence Elimination Step

The map file is divided into two parts. An partial example file for illustration is given in Fig. 1. The format is described based on this example file. Each occurring case is also covered in the description. The first line has to state "original variables" (line 1). This number is specified in the next line (line 2). Next, the compression information is given by beginning with either "compress table" (line 3), if there is a table, or "no table", if there is no compression. Afterwards, the tables are given where each starts with a line "table k v" and k represents the number of the table and v is the number of variables before the applied compression (line 4). The next line gives the com-

[1] The source code can be found at `www.ki.inf.tu-dresden.de/~norbert`.

```
 1:original variables
 2:30867
 3:compress tables
 4:table 0 30867
 5:1 2 3 5 6 7 9 10 11 ... 0
 6:units 0
 7:-31 32 ... -30666 -30822 0
 8:end table
 9:ee table
10:1 -19 0
11:2 -20 0
12:...
13:postprocess stack
14:ee
15:bce 523
16:-81 523 -6716 0
17:bce 10623
18:-10429 10623 -30296 0
19:...
20:ve 812 1
21:-812 -74 0
22:ve 6587 4
23:6587 6615 0
24:-79 6587 0
25:...
```

Fig. 1: Example map file

pression by simply giving a mapping that depends on the order: the i^{th} number in the line is the variable that is represented by variable i in the compressed formula (line 5). The line is closed by a 0, so that a standard clause parser can be used. The next line introduces the assignments in the original formula by saying "units k" (line 6). The following line lists all the literals that have been assigned *true* in the original formula and is also terminated by 0 (line 7). The compression is completed with a line stating "end table" (line 8). At the moment, only a single compression is supported, and thus, k is always 0. Since there is only a single compression, it is applied after applying all other techniques and therefore the following details are given with respect to the decompressed preprocessed formula F'. The next static information is the literals of the EE classes. They are introduced by a line "ee table" (line 9). The following lines represent the classes where the first element is the representative of the class that is in F'(line 10-12). Each class is ordered ascending, so that the EE information can be stored as a tree and the first element is the smallest one. Again, each class is terminated by a 0. Finally, the postprocess stack is given and preluded with a line "postprocess stack" (line 13). Afterwards the eliminations of BCE and VE are stored in the order they have been performed. BCE is prefaced with a line "bce l" where l is the blocking literal (line 15,17). The next line gives the according blocked clause (line 16,18). For VE the first line is "ve v n" where v is the eliminated variable and n is the number of clauses that have been replaced (line 20,22). The following n lines give the according clauses (line 21,23-26). Finally, for EE it is only stated that EE has been applied by writing a line "ee", because postprocessing EE depends also on the variables that are present at the moment (line 14). Some of the variables might already be removed at the point EE has been run, so that it is mandatory to store this information.

3.2 Preprocessor Comparison

A comparison of the formula reductions of *Coprocessor* and the current standard preprocessor SatELite is given in Fig. 2 and has been performed on 1155 industrial and crafted instances from recent SAT Competitions and SAT Races[2]. The relative reduction of the clauses by *Coprocessor* and SatELite is presented. Due to ER, *Coprocessor* can increase the number of clauses, whereby the average length is still reduced. *Coprocessor* is also able to reduce the number of clauses more than SatELite. The instances are ordered by the reduction of SatELite so that the plot for *Coprocessor* produces peaks.

Since SatELite [3] and *MiniSAT* [11] have been developed by the same authors, the run times of *MiniSAT* with the two preprocessors are compared in Fig. 3. Comparing these run times of *MiniSAT* (MS) combined with the preprocessors, it can be clearly seen that by using a preprocessor the performance of the solver is much higher. Furthermore, the combination with *Coprocessor* (MS+Co) solves more instances than *SatELite* (MS+S) for most of the timeouts.

[2] For more details visit `http://www.ki.inf.tu-dresden.de/~norbert/paperdata/WLP2011.html`.

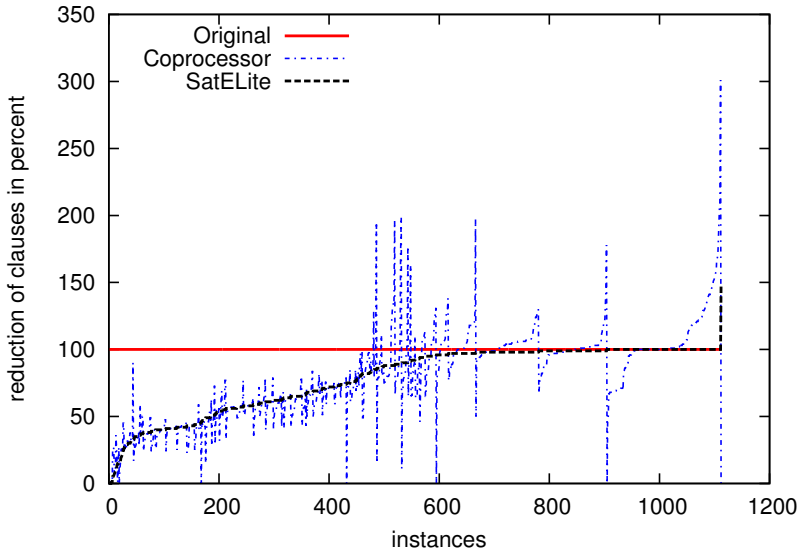


Fig. 2: Relative reduction of SatELite and Coprocessor

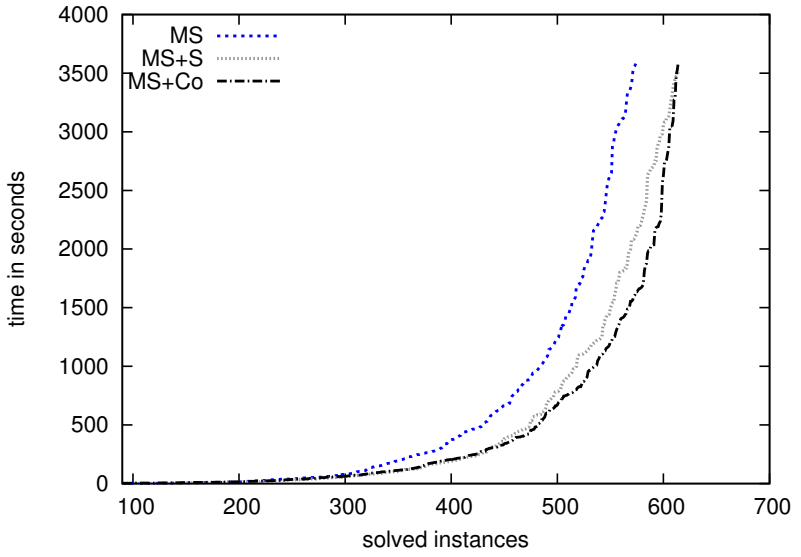


Fig. 3: Runtime comparison of MiniSAT combined with Coprocessor and SatELite

4 Conclusion and Future Work

This work introduces the SAT preprocessor *Coprocessor* that implements almost all known preprocessing techniques and some additional features. Experiments showed that the default *Coprocessor* performs better than *SatELite* when combined with *MiniSAT 2.2*. For suiting its techniques better to applications, *Coprocessor* provides many parameters that can be optimized for special use cases. Additionally, a map file format is presented that is used to store the preprocessing information. This file can be used to re-construct the model for the original formula if the model for the preprocessed formula is given.

Future development of this preprocessor includes adding the latest techniques such as HLE and HLA [4,5] and to parallelize it to be able to use multi-core architectures. Furthermore, the execution order of the techniques will be relaxed, so that any order can be applied to the input formula.

Acknowledgment The author would like to thank Marijn Heule for providing an implementation of HTE and discussing the dependencies of the algorithms.

References

1. G. Audemard, G. Katsirelos, and L. Simon. A restriction of extended resolution for clause learning sat solvers. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.
2. S. A. Cook. The complexity of theorem-proving procedures. In *Procs. 3rd Annual ACM Symposium on Theory of Computing*, 1971.
3. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.
4. M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for cnf formulas. In C. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.
5. M. Heule, M. Jarvisalo, and A. Biere. Efficient cnf simplification based on binary implication graphs. In K. Sakallah and L. Simon, editors, *SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, page 201215. Springer, 2011.
6. M. Järvisalo and A. Biere. Reconstructing solutions after blocked clause elimination. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 340–345. Springer Berlin / Heidelberg, 2010.
7. M. Järvisalo, A. Biere, and M. Heule. Blocked clause elimination. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin / Heidelberg, 2010.
8. I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pages 105–, Washington, DC, USA, 2003. IEEE Computer Society.
9. N. Manthey. Solver Submission of riss 1.0 to the SAT Competition 2011. Technical Report 1, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, Jan. 2011.
10. N. Manthey and P. Steinke. Quadratic Direct Encoding vs. Linear Order Encoding. Technical Report 3, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, June 2011.
11. Niklas Sörensson. Minisat 2.2 and minisat++ 1.1. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf`, 2010.
12. C. Piette, Y. Hamadi, and L. Sas. Vivifying propositional clausal formulae.
13. S. Subbarayan and D. K. Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer Berlin / Heidelberg, 2005.

Translating Answer-Set Programs into Bit-Vector Logic

Mai Nguyen, Tomi Janhunen, and Ilkka Niemelä

Aalto University School of Science
Department of Information and Computer Science
{Mai.Nguyen,Tomi.Janhunen,Ilkka.Niemela}@aalto.fi

Abstract. Answer set programming (ASP) is a paradigm for declarative problem solving where problems are first formalized as rule sets, i.e., answer-set programs, in a uniform way and then solved by computing answer sets for programs. The satisfiability modulo theories (SMT) framework follows a similar modelling philosophy but the syntax is based on extensions of propositional logic rather than rules. Quite recently, a translation from answer-set programs into difference logic was provided—enabling the use of particular SMT solvers for the computation of answer sets. In this paper, the translation is revised for another SMT fragment, namely that based on fixed-width bit-vector theories. Thus, even further SMT solvers can be harnessed for the task of computing answer sets. The results of a preliminary experimental comparison are also reported. They suggest a level of performance which is similar to that achieved via difference logic.

1 Introduction

Answer set programming (ASP) is a rule-based approach to declarative problem solving [15, 22, 24]. The idea is to first formalize a given problem as a set of rules also called an *answer-set program* so that the answer sets of the program correspond to the solution of the problem. Such problem descriptions are typically devised in a *uniform* way which distinguishes general principles and constraints of the problem in question from any instance-specific data. To this end, term variables are deployed for the sake of compact representation of rules. Solutions themselves can then be found out by *grounding* the rules of the answer-set program, and by computing answer sets for the resulting ground program using an answer set solver. State-of-the-art answer set solvers are already very efficient search engines [7, 11] and have a wide range of industrial applications.

The satisfiability modulo theories (SMT) framework [3] follows a similar modelling philosophy but the syntax is based on extensions of propositional logic rather than rules with term variables. The SMT framework enriches traditional satisfiability (SAT) checking [5] in terms of background theories which are selected amongst a number of alternatives.[1] Parallel to propositional atoms, also *theory atoms* involving non-Boolean variables[2] can be used as references to potentially infinite domains. Theory atoms are typically used to express various constraints such as linear constraints, difference constraints, etc., and they enable very concise representations of certain problem domains for which plain Boolean logic would be more verbose or insufficient in the first place.

As regards the relationship of ASP and SMT, it was quite recently shown [20, 25] that answer-set programs can be efficiently translated into a simple SMT fragment, namely *difference logic* (DL) [26]. This fragment is based on theory atoms of the form $x - y \leq k$ formalizing an upper bound k on the *difference* of two integer-domain variables x and y. Although the required transformation is linear, it is not reasonable to expect that such theories are directly written by humans in order to express the essentials of ASP in SMT. The translations from [20, 25] and their implementation called LP2DIFF[3] enable the use of particular SMT solvers for the computation of answer sets. Our experimental results [20] indicate that the performance obtained in this way is surprisingly close to that of state-of-the-art answer set solvers. The results of the third ASP competition [7], however, suggest that the performance gap has grown since the previous competition. To address this trend, our current and future agendas include a number of points:

[1] http://combination.cs.uiowa.edu/smtlib/

[2] However, variables in SMT are syntactically represented by (functional) constants having a free interpretation over a specific domain such as integers or reals.

[3] http://www.tcs.hut.fi/Software/lp2diff/

- We gradually increase the number of supported SMT fragments which enables the use of further SMT solvers for the task of computing answer sets.
- We continue the development of new translation techniques from ASP to SMT.
- We submit ASP-based benchmark sets to future SMT competitions (SMT-COMPs) to foster the efficiency of SMT solvers on problems that are relevant for ASP.
- We develop new integrated languages that combine features of ASP and SMT, and aim at implementations via translation into pure SMT as initiated in [18].

This paper contributes to the first item by devising a translation from answer-set programs into theories of bit-vector logic. There is a great interest to develop efficient solvers for this particular SMT fragment due to its industrial relevance. In view of the second item, we generalize an existing translation from [20] to the case of bit-vector logic. Using an implementation of the new translation, viz. LP2BV, new benchmark classes can be created to support the third item on our agenda. Finally, the translation also creates new potential for language integration. In the long run, rule-based languages and, in particular, the modern grounders exploited in ASP can provide valuable machinery for the generation of SMT theories in analogy to answer-set programs: The *source code* of an SMT theory can be compacted using rules and term variables [18] and specified in a uniform way which is independent of any concrete problem instances. Analogous approaches [2, 14, 23] combine ASP and constraint programming techniques without a translation.

The rest of this paper is organized as follows. First, the basic definitions and concepts of answer-set programs and fixed-width bit-vector logic are briefly reviewed in Section 2. The new translation from answer-set programs into bit-vector theories is then devised in Section 3. The extended rule types of SMODELS compatible systems are addressed in Section 4. Such extensions can be covered either by native translations into bit-vector logic or translations into normal programs. As part of this research, we carried out a number of experiments using benchmarks from the second ASP competition [11] and two state-of-the-art SMT solvers, viz. BOOLECTOR and Z3. The results of the experiments are reported in Section 5. Finally, we conclude this paper in Section 6 in terms of discussions of results and future work.

2 Preliminaries

The goal of this section is to briefly review the source and target formalisms for the new translation devised in the sequel. First, in Section 2.1, we recall normal logic programs subject to answer set semantics and the main notions exploited in their translation. A formal account of bit-vector logic follows in Section 2.2.

2.1 Normal Logic Programs

As usual, we define a *normal logic program* P as a finite set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m \tag{1}$$

where a, $b_1, \ldots, b_n$, and $c_1, \ldots, c_m$ are propositional atoms and $\sim$ denotes *default negation*. The *head* of a rule r of the form (1) is $\mathrm{hd}(r) = a$ whereas the part after the symbol $\leftarrow$ forms the *body* of r, denoted by $\mathrm{bd}(r)$. The body $\mathrm{bd}(r)$ consists of the positive part $\mathrm{bd}^+(r) = \{b_1, \ldots, b_n\}$ and the negative part $\mathrm{bd}^-(r) = \{c_1, \ldots, c_m\}$ so that $\mathrm{bd}(r) = \mathrm{bd}^+(r) \cup \{{\sim}c \mid c \in \mathrm{bd}^-(r)\}$. Intuitively, a rule r of the form (1) appearing in a program P is used as follows: the head $\mathrm{hd}(r)$ can be inferred by r if the *positive body atoms* in $\mathrm{bd}^+(r)$ are inferable by the other rules of P, but not the *negative body atoms* in $\mathrm{bd}^-(r)$. The positive part of the rule, r^+ is defined as $\mathrm{hd}(r) \leftarrow \mathrm{bd}^+(r)$. A normal logic program is called *positive* if $r = r^+$ holds for every rule $r \in P$.

Semantics To define the semantics of a normal program P, we let $\mathrm{At}(P)$ stand for the set of atoms that appear in P. An *interpretation* of P is any subset $I \subseteq \mathrm{At}(P)$ such that for an atom $a \in \mathrm{At}(P)$, a is *true* in I, denoted $I \models a$, iff $a \in I$. For any negative literal ${\sim}c$, $I \models {\sim}c$ iff $I \not\models c$ iff $c \notin I$. A rule r is satisfied in I, denoted $I \models r$, iff $I \models \mathrm{bd}(r)$ implies $I \models \mathrm{hd}(r)$. An interpretation I is a *classical model* of P, denoted $I \models P$, iff, $I \models r$ holds for every $r \in P$. A model $M \models P$ is a *minimal model* of P iff there is no $M' \models P$ such that $M' \subset M$. Each positive normal program P has a unique minimal model, i.e., the *least model*

of P denoted by $\mathrm{LM}(P)$ in the sequel. The least model semantics can be extended for an arbitrary normal program P by *reducing* P into a positive program $P^M = \{r^+ \mid r \in P \text{ and } M \cap \mathrm{bd}^-(r) = \emptyset\}$ with respect to $M \subseteq \mathrm{At}(P)$. Then *answer sets*, also known as *stable models* [16], can be defined.

Definition 1 (Gelfond and Lifschitz [16]). *An interpretation $M \subseteq \mathrm{At}(P)$ is an* answer set *of a normal program P iff $M = \mathrm{LM}(P^M)$.*

Example 1. Consider a normal program P [20] consisting of the following six rules:

$$\begin{array}{lll} a \leftarrow b, c. & a \leftarrow d. & b \leftarrow a, {\sim}d. \\ b \leftarrow a, {\sim}c. & c \leftarrow {\sim}d. & d \leftarrow {\sim}c. \end{array}$$

The answer sets of P are $M_1 = \{a, b, d\}$ and $M_2 = \{c\}$. To verify the latter, we note that $P^{M_2} = \{a \leftarrow b, c;\ b \leftarrow a;\ c \leftarrow;\ a \leftarrow d\}$ for which $\mathrm{LM}(P^{M_2}) = \{c\}$. On the other hand, we have $P^{M_3} = P^{M_2}$ for $M_3 = \{a, b, c\}$ so that $M_3 \notin \mathrm{AS}(P)$. ■

The number of answer sets possessed by a normal program P can vary in general. The set of answer sets of a normal program P is denoted by $\mathrm{AS}(P)$. Next we present some concepts and results that are relevant in order to capture answer sets in terms of propositional logic and its extensions in the SMT framework.

Completion Given a normal program P and an atom $a \in \mathrm{At}(P)$, the *definition* of a in P is the set of rules $\mathrm{Def}_P(a) = \{r \in P \mid \mathrm{hd}(r) = a\}$. The *completion* of a normal program P, denoted by $\mathrm{Comp}(P)$, is a propositional theory [8] which contains

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \Big(\bigwedge_{b \in \mathrm{bd}^+(r)} b \ \wedge \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c \Big) \tag{2}$$

for each atom $a \in \mathrm{At}(P)$. Given a propositional theory T and its signature $\mathrm{At}(T)$, the semantics of T is determined by $\mathrm{CM}(T) = \{M \subseteq \mathrm{At}(T) \mid M \models T\}$. It is possible to relate $\mathrm{CM}(\mathrm{Comp}(P))$ with the models of a normal program P by distinguishing *supported models* [1] for P. A model $M \models P$ is a supported model of P iff for every atom $a \in M$ there is a rule $r \in P$ such that $\mathrm{hd}(r) = a$ and $M \models \mathrm{bd}(r)$. In general, the set of supported models $\mathrm{SuppM}(P)$ of a normal program P coincides with $\mathrm{CM}(\mathrm{Comp}(P))$. It can be shown [21] that stable models are also supported models but not necessarily vice versa. This means that in order to capture $\mathrm{AS}(P)$ using $\mathrm{Comp}(P)$, the latter has to be extended in terms of additional constraints as done, e.g., in [17, 20].

Example 2. For the program P of Example 1, the theory $\mathrm{Comp}(P)$ has formulas $a \leftrightarrow (b \wedge c) \vee d$, $b \leftrightarrow (a \wedge \neg d) \vee (a \wedge \neg c)$, $c \leftrightarrow \neg d$, and $d \leftrightarrow \neg c$. The models of $\mathrm{Comp}(P)$, i.e., its supported models, are $M_1 = \{a, b, d\}$, $M_2 = \{c\}$, and $M_3 = \{a, b, c\}$. ■

Dependency Graphs The *positive dependency graph* of a normal program P, denoted by $\mathrm{DG}^+(P)$, is a pair $\langle \mathrm{At}(P), \leq \rangle$ where $b \leq a$ holds iff there is a rule $r \in P$ such that $\mathrm{hd}(r) = a$ and $b \in \mathrm{bd}^+(r)$. Let $\leq^*$ denote the *reflexive* and *transitive* closure of $\leq$. A *strongly connected component* (SCC) of $\mathrm{DG}^+(P)$ is a maximal non-empty subset $S \subseteq \mathrm{At}(P)$ such that $a \leq^* b$ and $b \leq^* a$ hold for each $a, b \in S$. The set of defining rules is generalized for an SCC S by $\mathrm{Def}_P(S) = \bigcup_{a \in S} \mathrm{Def}_P(a)$. This set can be naturally partitioned into sets $\mathrm{Ext}_P(S) = \{r \in \mathrm{Def}_P(S) \mid \mathrm{bd}^+(r) \cap S = \emptyset\}$ and $\mathrm{Int}_P(S) = \{r \in \mathrm{Def}_P(S) \mid \mathrm{bd}^+(r) \cap S \neq \emptyset\}$ of *external* and *internal* rules associated with S, respectively. Thus, $\mathrm{Def}_P(S) = \mathrm{Ext}_P(S) \sqcup \mathrm{Int}_P(S)$ holds in general.

Example 3. In the case of the program P from Example 1, the SCCs of $\mathrm{DG}^+(P)$ are $S_1 = \{a, b\}$, $S_2 = \{c\}$, and $S_3 = \{d\}$. For S_1, we have $\mathrm{Ext}_P(S_1) = \{a \leftarrow d\}$. ■

2.2 Bit-Vector Logic

Fixed-width bit-vector theories have been introduced for high-level reasoning about digital circuitry and computer programs in the SMT framework [27, 4]. Such theories are expressed in an extension of propositional logic where atomic formulas speak about bit vectors in terms of a rich variety of operators.

Syntax As usual in the context of SMT, variables are realized as constants that have a free interpretation over a particular domain (such as integers or reals)[4]. In the case of fixed-width bit-vector theories, this means that each constant symbol x represents a vector $x[1 \dots m]$ of bits of particular width m, denoted by $\mathrm{w}(x)$ in the sequel. Such vectors enable a more compact representation of structures like registers and often allow more efficient reasoning about them. A special notation $\overline{n}$ is introduced to denote a bit vector that equals to n, i.e., $\overline{n}$ provides a binary representation of n. We assume that the actual width $m \geq \log_2(n+1)$ is determined by the context where the notation $\overline{n}$ is used. For the purposes of this paper, the most interesting arithmetic operator for combining bit vectors is the addition of two m-bit vectors, denoted by the parameterized function symbol $+_m$ in an infix notation. The resulting vector is also m-bit which can lead to an overflow if the sum exceeds $2^m - 1$. Moreover, we use Boolean operators $=_m$ and $<_m$ with the usual meanings for comparing the values of two m-bit vectors. Thus, assuming that x and y are m-bit free constants, we may write atomic formulas like $x =_m y$ and $x <_m y$ in order to compare the m-bit values of x and y. In addition to syntactic elements mentioned so far, we can use the primitives of propositional logic to build more complex *well-formed formulas* of bit-vector logic. The syntax defined for the SMT library contains further primitives which are skipped in this paper. A theory T in bit-vector logic is a set of well-formed bit-vector formulas as illustrated by the following example.

Example 4. Consider a system of two processes, say A and B, and a theory $T = \{a \rightarrow (x <_2 y),\ b \rightarrow (y <_2 x)\}$ formalizing a scheduling policy for them. The intuitive reading of a (resp. b) is that process A (resp. B) is scheduled with a higher priority and, thus, should start earlier. The constants x and y denote the respective starting times of A and B. Thus, e.g., $x <_2 y$ means that process A starts before process B. ■

Semantics Given a bit-vector theory T, we write $\mathrm{At}(T)$ and $\mathrm{FC}(T)$ for the sets of propositional atoms and free constants, respectively, appearing in T. To determine the semantics of T, we define *interpretations* for T as pairs $\langle I, \tau \rangle$ where $I \subseteq \mathrm{At}(T)$ is a standard propositional interpretation and τ is a partial function that maps a free constant $x \in \mathrm{FC}(T)$ and an index $1 \leq i \leq \mathrm{w}(x)$ to the set of bits $\{0, 1\}$. Given τ, a constant $x \in \mathrm{FC}(T)$ is mapped onto $\tau(x) = \sum_{i=1}^{\mathrm{w}(x)} (\tau(x, i) \cdot 2^{\mathrm{w}(x)-i})$ and, in particular, $\tau(\overline{n}) = n$ for any n. To cover any *well-formed terms*[5] t_1 and t_2 involving $+_m$ and m-bit constants from $\mathrm{FC}(T)$, we define $\tau(t_1 +_m t_2) = \tau(t_1) + \tau(t_2) \mod 2^m$ and $\mathrm{w}(t_1 +_m t_2) = m$. Hence, the value $\tau(t)$ can be determined for any well-formed term t which enables the evaluation of more complex formulas as formalized below.

Definition 2. *Let T be a bit-vector theory, $a \in \mathrm{At}(T)$ a propositional atom, t_1 and t_2 well-formed terms over $\mathrm{FC}(T)$ such that $\mathrm{w}(t_1) = \mathrm{w}(t_2)$, and ϕ and ψ well-formed formulas. Given an interpretation $\langle I, \tau \rangle$ for the theory T, we define*

1. $\langle I, \tau \rangle \models a \iff a \in I$,
2. $\langle I, \tau \rangle \models t_1 =_m t_2 \iff \tau(t_1) = \tau(t_2)$,
3. $\langle I, \tau \rangle \models t_1 <_m t_2 \iff \tau(t_1) < \tau(t_2)$,
4. $\langle I, \tau \rangle \models \neg\phi \iff \langle I, \tau \rangle \not\models \phi$,
5. $\langle I, \tau \rangle \models \phi \vee \psi \iff \langle I, \tau \rangle \models \phi$ *or* $\langle I, \tau \rangle \models \psi$,
6. $\langle I, \tau \rangle \models \phi \rightarrow \psi \iff \langle I, \tau \rangle \not\models \phi$ *or* $\langle I, \tau \rangle \models \psi$, *and*
7. $\langle I, \tau \rangle \models \phi \leftrightarrow \psi \iff \langle I, \tau \rangle \models \phi$ *if and only if* $\langle I, \tau \rangle \models \psi$.

The interpretation $\langle I, \tau \rangle$ is a model of T, i.e., $\langle I, \tau \rangle \models T$, iff $\langle I, \tau \rangle \models \phi$ for all $\phi \in T$.

It is clear by Definition 2 that pure propositional theories T are treated classically, i.e., $\langle I, \tau \rangle \models T$ iff $I \models T$ in the sense of propositional logic. As regards the theory T from Example 4, we have the sets of symbols $\mathrm{At}(T) = \{a, b\}$ and $\mathrm{FC}(T) = \{x, y\}$. Furthermore, we observe that there is no model of T of the form $\langle \{a, b\}, \tau \rangle$ because it is impossible to satisfy $x <_2 y$ and $y <_2 x$ simultaneously using any partial function τ. On the other hand, there are 6 models of the form $\langle \{a\}, \tau \rangle$ because $x <_2 y$ can be satisfied in $3 + 2 + 1 = 6$ ways by picking different values for the 2-bit vectors x and y.

[4] We use typically symbols x, y, z to denote such free (functional) constants and symbols a, b, c to denote propositional atoms.

[5] The constants and operators appearing in a well-formed term t are based on a fixed width m. Moreover, the width $\mathrm{w}(x)$ of each constant $x \in \mathrm{FC}(T)$ must be the same throughout T.

3 Translation

In this section, we present a translation of a logic program P into a bit-vector theory $\mathrm{BV}(P)$ that is similar to an existing translation [20] into difference logic. As its predecessor, the translation $\mathrm{BV}(P)$ consists of two parts. Clark's completion [8], denoted by $\mathrm{CC}(P)$, forms the first part of $\mathrm{BV}(P)$. The second part, i.e., $\mathrm{R}(P)$, is based on *ranking constraints* from [25] so that $\mathrm{BV}(P) = \mathrm{CC}(P) \cup \mathrm{R}(P)$. Intuitively, the idea is that the completion $\mathrm{CC}(P)$ captures *supported models* of P [1] and the further formulas in $\mathrm{R}(P)$ exclude the non-stable ones so that any classical model of $\mathrm{BV}(P)$ corresponds to a stable model of P.

The completion $\mathrm{CC}(P)$ is formed for each atom $a \in \mathrm{At}(P)$ on the basis of (2):

1. If $\mathrm{Def}_P(a) = \emptyset$, the formula $\neg a$ is included to capture the corresponding empty disjunction in (2).
2. If there is $r \in \mathrm{Def}_P(a)$ such that $\mathrm{bd}(r) = \emptyset$, then one of the disjuncts in (2) is trivially true and the formula a can be used as such to capture the definition of a.
3. If $\mathrm{Def}_P(a) = \{r\}$ for a rule $r \in P$ with $n + m > 0$, then we simplify (2) to a formula of the form
$$a \leftrightarrow \bigwedge_{b \in \mathrm{bd}^+(r)} b \;\wedge\; \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c. \tag{3}$$
4. Otherwise, the set $\mathrm{Def}_P(a)$ contains at least two rules (1) with $n + m > 0$ and
$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \mathrm{bd}_r \tag{4}$$
is introduced using a new atom bd_r for each $r \in \mathrm{Def}_P(a)$ together with a formula
$$\mathrm{bd}_r \leftrightarrow \bigwedge_{b \in \mathrm{bd}^+(r)} b \;\wedge\; \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c. \tag{5}$$

The rest of the translation exploits the SCCs of the positive dependency graph of P that was defined in Section 2.1. The motivation is to limit the scope of ranking constraints which favors the length of the resulting translation. In particular, singleton components $\mathrm{SCC}(a) = \{a\}$ require no special treatment if *tautological* rules with $a \in \{b_1, \ldots, b_n\}$ in (1) have been removed. Plain completion (2) is sufficient for atoms involved in such components. However, for each atom $a \in \mathrm{At}(P)$ having a non-trivial component $\mathrm{SCC}(a)$ in $\mathrm{DG}^+(P)$ such that $|\mathrm{SCC}(a)| > 1$, two new atoms ext_a and int_a are introduced to formalize the *external* and *internal* support for a, respectively. These atoms are defined in terms of equivalences

$$\mathrm{ext}_a \leftrightarrow \bigvee_{r \in \mathrm{Ext}_P(a)} \mathrm{bd}_r \tag{6}$$

$$\mathrm{int}_a \leftrightarrow \bigvee_{r \in \mathrm{Int}_P(a)} \Big[\mathrm{bd}_r \wedge \bigwedge_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} (x_b <_m x_a)\Big] \tag{7}$$

where x_a and x_b are bit vectors of width $m = \lceil \log_2(|\mathrm{SCC}(a)| + 1) \rceil$ introduced for all atoms involved in $\mathrm{SCC}(a)$. The formulas (6) and (7) are called *weak* ranking constraints and they are accompanied by

$$a \rightarrow \mathrm{ext}_a \vee \mathrm{int}_a, \tag{8}$$

$$\neg \mathrm{ext}_a \vee \neg \mathrm{int}_a. \tag{9}$$

Moreover, when $\mathrm{Ext}_P(a) \neq \emptyset$ and the atom a happens to gain external support from these rules, the value of x_a is fixed to 0 by including the formula

$$\mathrm{ext}_a \rightarrow (x_a =_m \overline{0}). \tag{10}$$

Example 5. Recall the program P from Example 1. The completion $\mathrm{CC}(P)$ is:

$$\begin{array}{lll} a \leftrightarrow \mathrm{bd}_1 \vee \mathrm{bd}_2. & \mathrm{bd}_1 \leftrightarrow b \wedge c. & \mathrm{bd}_2 \leftrightarrow d. \\ b \leftrightarrow \mathrm{bd}_3 \vee \mathrm{bd}_4. & \mathrm{bd}_3 \leftrightarrow a \wedge \neg d. & \mathrm{bd}_4 \leftrightarrow a \wedge \neg c. \\ c \leftrightarrow \neg d. & & \\ d \leftrightarrow \neg c. & & \end{array}$$

Since P has only one non-trivial SCC, i.e., the component $\mathrm{SCC}(a) = \mathrm{SCC}(b) = \{a, b\}$, the weak ranking constraints resulting in $\mathrm{R}(P)$ are

$$\begin{array}{l} \mathrm{ext}_a \leftrightarrow \mathrm{bd}_2. \quad \mathrm{int}_a \leftrightarrow \mathrm{bd}_1 \wedge (x_b <_2 x_a). \\ \mathrm{ext}_b \leftrightarrow \bot\ . \\ \mathrm{int}_b \leftrightarrow [\mathrm{bd}_3 \wedge (x_a <_2 x_b)] \vee [\mathrm{bd}_4 \wedge (x_a <_2 x_b)]. \end{array}$$

In addition to these, the formulas

$$\begin{array}{llll} a \rightarrow \mathrm{ext}_a \vee \mathrm{int}_a. & \neg\mathrm{ext}_a \vee \neg\mathrm{int}_a. & \mathrm{ext}_a \rightarrow (x_a =_2 \overline{0}). \\ b \rightarrow \mathrm{ext}_b \vee \mathrm{int}_b. & \neg\mathrm{ext}_b \vee \neg\mathrm{int}_b. \end{array}$$

are also included in $\mathrm{R}(P)$. ■

Weak ranking constraints are sufficient whenever the goal is to compute only one answer set, or to check the existence of answer sets. However, they do not guarantee a one-to-one correspondence between the elements of $\mathrm{AS}(P)$ and the set of models obtained for the translation $\mathrm{BV}(P)$. To address this discrepancy, and to potentially make the computation of all answer sets or counting the number of answer sets more effective, *strong* ranking constraints can be imported from [20] as well. Actually, there are two mutually compatible variants of strong ranking constraints:

$$\mathrm{bd}_r \rightarrow \bigvee_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} \neg(x_b +_m \overline{1} <_m x_a) \tag{11}$$

$$\mathrm{int}_a \rightarrow \bigvee_{r \in \mathrm{Int}_P(a)} [\mathrm{bd}_r \wedge \bigvee_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} (x_a =_m x_b +_m \overline{1})]. \tag{12}$$

The *local* strong ranking constraint (11) is introduced for each $r \in \mathrm{Int}_P(a)$. It is worth pointing out that the condition $\neg(x_b +_m \overline{1} <_m x_a)$ is equivalent to $x_b +_m \overline{1} \geq_m x_a$. [6] On the other hand, the *global* variant (12) covers the internal support of a entirely. Finally, in order to prune copies of models of the translation that would correspond to the exactly same answer set of the original program, a formula

$$\neg a \rightarrow (x_a =_m \overline{0}) \tag{13}$$

is included for every atom a involved in a non-trivial SCC. We write $\mathrm{R}^{\mathrm{l}}(P)$ and $\mathrm{R}^{\mathrm{g}}(P)$ for the respective extensions of $\mathrm{R}(P)$ with local/global strong ranking constraints, and $\mathrm{R}^{\mathrm{lg}}(P)$ obtained using both. Similar conventions are applied to $\mathrm{BV}(P)$ to distinguish four variants in total. The correctness of these translations is addressed next.

Theorem 1. *Let P be a normal program and $\mathrm{BV}(P)$ its bit-vector translation.*

1. *If S is an answer set of P, then there is a model $\langle M, \tau \rangle$ of $\mathrm{BV}(P)$ such that $S = M \cap \mathrm{At}(P)$.*
2. *If $\langle M, \tau \rangle$ is a model of $\mathrm{BV}(P)$, then $S = M \cap \mathrm{At}(P)$ is an answer set of P.*

Proof. To establish the correspondence of answer sets and models as formalized above, we appeal to the analogous property of the translation of P into difference logic (DL), denoted here by $\mathrm{DL}(P)$. In DL, theory atoms $x \leq y + k$ constrain the difference of two integer variables x and y. Models can be represented as pairs $\langle I, \tau \rangle$ where I is a propositional interpretation and τ maps constants of theory atoms to integers so that $\langle I, \tau \rangle \models x \leq y + k \iff \tau(x) \leq \tau(y) + k$. The rest is analogous to Definition 2.

($\Longrightarrow$) Suppose that S is an answer set of P. Then the results of [20] imply that there is a model $\langle M, \tau \rangle$ of $\mathrm{DL}(P)$ such that $S = M \cap \mathrm{At}(P)$. The valuation τ is condensed for each non-trivial SCC S of $\mathrm{DG}^+(P)$ as follows. Let us partition S into $S_0 \sqcup \ldots \sqcup S_n$ such that (i) $\tau(x_a) = \tau(x_b)$ for each $0 \leq i \leq n$ and $a, b \in S_i$, (ii) $\tau(x_a) = \tau(z)$[7] for each $a \in S_0$, and (iii) for each $0 \leq i < j \leq n$, $a \in S_i$, and $b \in S_j$, $\tau(x_a) \leq \tau(x_b)$. Then define τ' for the bit vector x_a associated with an atom $a \in S_i$ by setting $\tau'(x_a, j) = 1$ iff the j^{th} bit of $\bar{i}$ is 1, i.e., $\tau'(x_a) = i$. It follows that $\langle I, \tau \rangle \models x_b \leq x_a - 1$ iff $\langle I, \tau' \rangle \models x_b <_m x_a$ for

[6] However, the form in (11) is used in our implementation, since $+_m$ and $<_m$ are amongst the base operators of the BOOLECTOR system.

[7] A special variable z is used as a placeholder for the constant 0 in the translation $\mathrm{DL}(P)$ [20].

any $a, b \in S$. Moreover, we have $\langle M, \tau\rangle \models (x_a \leq z + 0) \wedge (z \leq x_a + 0)$ iff $\langle M, \tau'\rangle \models x_a =_m \overline{0}$ for any $a \in S$. Due to the similar structures of $\mathrm{DL}(P)$ and $\mathrm{BV}(P)$, we obtain $\langle M, \tau\rangle \models \mathrm{BV}(P)$ as desired.

($\Longleftarrow$) Let $\langle M, \tau\rangle$ be a model of $\mathrm{BV}(P)$. Then define τ' such that $\tau'(x) = \sum_{i=1}^{\mathrm{w}(x)} (\tau(x, i) \cdot 2^{\mathrm{w}(x)-i})$ where x on the left hand side stands for the integer variable corresponding to the bit vector x on the right hand side. It follows that $\langle I, \tau\rangle \models x_b <_m x_a$ iff $\langle I, \tau'\rangle \models x_b \leq x_a - 1$. By setting $\tau'(z) = 0$, we obtain $\langle M, \tau\rangle \models x_a =_m \overline{0}$ if and only if $\langle M, \tau'\rangle \models (x_a \leq z + 0) \wedge (z \leq x_a + 0)$. The strong analogy present in the structures of $\mathrm{BV}(P)$ and $\mathrm{DL}(P)$ implies that $\langle M, \tau'\rangle$ is a model of $\mathrm{DL}(P)$. Thus, $S = M \cap \mathrm{At}(P)$ is an answer set of P by [20]. □

Even tighter relationships of answer sets and models can be established for the translations $\mathrm{BV}^{\mathrm{l}}(P)$, $\mathrm{BV}^{\mathrm{g}}(P)$, and $\mathrm{BV}^{\mathrm{lg}}(P)$. It can be shown that the model $\langle M, \tau\rangle$ of $\mathrm{BV}^*(P)$ corresponding to an answer set S of P is unique, i.e., there is no other model $\langle N, \tau'\rangle$ of the translation such that $S = N \cap \mathrm{At}(P)$. These results contrast with [20]: the analogous extensions $\mathrm{DL}^*(P)$ guarantee the uniqueness of M in a model $\langle M, \tau\rangle$ but there are always infinitely many copies $\langle M, \tau'\rangle$ of $\langle M, \tau\rangle$ such that $\langle M, \tau'\rangle \models \mathrm{DL}^*(P)$. Such a valuation τ' can be simply obtained by setting $\tau'(x) = \tau(x) + 1$ for any x.

4 Native Support for Extended Rule Types

The input syntax of the SMODELS system was soon extended by further rule types [28]. In solver interfaces, the rule types usually take the following simple syntactic forms:

$$\{a_1, \ldots, a_l\} \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m. \tag{14}$$

$$a \leftarrow l\{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\}. \tag{15}$$

$$a \leftarrow l\{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \sim c_1 = w_{c_1}, \ldots, \sim c_m = w_{c_m}\}. \tag{16}$$

The body of a *choice rule* (14) is interpreted in the same way as that of a normal rule (1). The head, in contrast, allows to derive any subset of atoms $a_1, \ldots, a_l$, if the body is satisfied, and to make a *choice* in this way. The head a of a *cardinality rule* (15) is derived, if its body is satisfied, i.e., the number of satisfied literals amongst $b_1, \ldots, b_n$ and $\sim c_1, \ldots, \sim c_m$ is at least l acting as the *lower bound*. A *weight rule* of the form (16) generalizes this idea by assigning arbitrary positive weights to literals (rather than 1s). The body is satisfied if the sum of weights assigned to satisfied literals is at least l, thus enabling one to infer the head a using the rule. In practise, the grounding components used in ASP systems allow for more versatile use of cardinality and weight rules, but the primitive forms (14), (15), and (16) provide a solid basis for efficient implementation via translations. The reader is referred to [28] for a generalization of answer sets for programs involving such extended rule types. The respective class of *weight constraint programs* (WCPs) is typically supported by SMODELS compatible systems.

Whenever appropriate, it is possible to translate extended rule types as introduced above back to normal rules. To this end, a number of transformations are addressed in [19] and they have been implemented as a tool called LP2NORMAL[8]. For instance, the head of a choice rule (14) can be captured in terms of rules

$$\begin{array}{llll} a_1 \leftarrow b, \sim\overline{a_1}. & \ldots & a_l \leftarrow b, \sim\overline{a_l}. \\ \overline{a_1} \leftarrow \sim a_1. & \ldots & \overline{a_l} \leftarrow \sim a_l. \end{array}$$

where $\overline{a_1}, \ldots, \overline{a_l}$ are new atoms and b is a new atom standing for the body of (14) which can be defined using (14) with the head replaced by b. We assume that this transformation is applied at first to remove choice rules when the goal is to translate extended rule types into bit-vector logic. The strength of this transformation is locality, i.e., it can be applied on a rule-by-rule basis, and linearity with respect to the length of the original rule (14). To the contrary, linear normalization of cardinality and weight rules seems impossible. Thus, we also provide direct translations into formulas of bit-vector logic.

We present the translation of a weight rule (16) whereas the translation of a cardinality rule (15) is obtained as a special case $w_{b_1} = \ldots = w_{b_n} = w_{c_1} = \ldots = w_{c_m} = 1$. The body of a weight rule can be evaluated using bit vectors $s_1, \ldots, s_{n+m}$ of width $k = \lceil \log_2(\sum_{i=1}^{n} w_{b_i} + \sum_{i=1}^{m} w_{c_i} + 1)\rceil$ constrained by $2 \times (n + m)$ formulas

[8] http://www.tcs.hut.fi/Software/asptools/

```
gringo program.lp instance.lp \
| smodels -internal -nolookahead \
| lpcat -s=symbols.txt \
| lp2bv [-l] [-g] \
| boolector -fm
```

Fig. 1. Unix shell pipeline for running a benchmark instance

$$
\begin{array}{ll}
b_1 \rightarrow (s_1 =_k \overline{w_{b_1}}), & \neg b_1 \rightarrow (s_1 =_k \overline{0}), \\
b_2 \rightarrow (s_2 =_k s_1 +_k \overline{w_{b_2}}), & \neg b_2 \rightarrow (s_2 =_k s_1), \\
\vdots & \vdots \\
b_n \rightarrow (s_n =_k s_{n-1} +_k \overline{w_{b_n}}), & \neg b_n \rightarrow (s_n =_k s_{n-1}), \\
c_1 \rightarrow (s_{n+1} =_k s_n), & \neg c_1 \rightarrow (s_{n+1} =_k s_n +_k \overline{w_{c_1}}), \\
\vdots & \vdots \\
c_m \rightarrow (s_{n+m} =_k s_{n+m-1}), & \neg c_m \rightarrow (s_{n+m} =_k s_{n+m-1} +_k \overline{w_{c_m}}).
\end{array}
$$

The lower bound l of (16) can be checked in terms of the formula $\neg(s_{n+m} <_k \bar{l})$ where we assume that $\bar{l}$ is of width k, since the rule can be safely deleted otherwise. In view of the overall translation, the formula $\mathrm{bd}_r \leftrightarrow \neg(s_{n+m} <_k \bar{l})$ can be used in conjunction with the completion formula (4). Weight rules also contribute to the dependency graph $\mathrm{DG}^+(P)$ in analogy to normal rules, i.e., the head a depends on all positive body atoms $b_1, \ldots, b_n$. In this way, $\mathrm{BV}(P)$ generalizes for programs P having extended rules.

5 Experimental Results

A new translator called LP2BV was implemented as a derivative of LP2DIFF[9] that translates logic programs into difference logic. In contrast, the new translator will provide its output in the bit-vector format. In analogy to its predecessor, it expects to receive its input in the SMODELS[10] file format. Models of the resulting bit-vector theory are searched for using BOOLECTOR[11] (v. 1.4.1) [6] and Z3[12] (v. 2.11) [9] as back-end solvers. The goal of our preliminary experiments was to see how the performances of systems based on LP2BV compare with the performance of a state-of-the-art ASP solver CLASP[13] (v. 1.3.5) [13]. The experiments were based on the NP-complete benchmarks of the ASP Competition 2009. In this benchmark collection, there are 23 benchmark problems with 516 instances in total. Before invoking a translator and the respective SMT solver, we performed a few preprocessing steps, as detailed in Figure 1, by calling:

- GRINGO (v. 2.0.5), for grounding the problem encoding and a given instance;
- SMODELS[14] (v. 2.34), for simplifying the resulting ground program;
- LPCAT (v. 1.18), for removing all unused atom numbers, for making the atom table of the ground program contiguous, and for extracting the symbols for later use; and
- LP2NORMAL (version 1.11), for normalizing the program.

The last step is optional and not included as part of the pipeline in Figure 1. Pipelines of this kind were executed under Linux/Ubuntu operating system running on six-core AMD Opteron$^{(\mathrm{TM})}$ 2435 processors under 2.6 GHz clock rate and with 2.7 GB memory limit that corresponds to the amount of memory available in the ASP Competition 2009.

For each system based on a translator and a back-end solver, there are four variants of the system to consider: W indicates that only weak ranking constraints are used, while L, G, and LG mean that either local, or global, or both local and global strong ranking constraints, respectively, are employed when translating the logic program.

[9] http://www.tcs.hut.fi/Software/lp2diff/
[10] http://www.tcs.hut.fi/Software/smodels/
[11] http://fmv.jku.at/boolector/
[12] http://research.microsoft.com/en-us/um/redmond/projects/z3/
[13] http://www.cs.uni-potsdam.de/clasp/
[14] http://www.tcs.hut.fi/Software/smodels/

Table 1. Experimental results without normalization

Benchmark	INST	CLASP	LP2BV+BOOLECTOR				LP2BV+Z3				LP2DIFF+Z3			
			W	L	G	LG	W	L	G	LG	W	L	G	LG
Overall Performance	516	465	276	244	261	256	217	216	194	204	**360**	349	324	324
		347/118	188/ 88	161/ 83	174/ 87	176/ 80	142/ 75	147/ 69	124/ 70	135/ 69	**257/103**	251/ 98	225/ 99	226/ 98
KnightTour	10	8/ 0	2/ 0	1/ 0	0/ 0	0/ 0	1/ 0	0/ 0	0/ 0	1/ 0	**6/ 0**	**6/ 0**	4/ 0	5/ 0
GraphColouring	29	8/ 0	**7/0**	**7/0**	**7/0**	**7/0**	6/ 0	**7/0**	**7/0**	**7/0**	**7/0**	**7/0**	**7/0**	**7/0**
WireRouting	23	11/11	2/ 3	1/ 1	1/ 2	0/ 2	1/ 3	0/ 0	0/ 0	0/ 1	3/ 3	2/ 3	2/ 4	**5/3**
DisjunctiveScheduling	10	5/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0
GraphPartitioning	13	6/ 7	3/ 0	3/ 0	3/ 0	3/ 0	4/ 0	4/ 0	4/ 0	3/ 0	**6/2**	6/ 1	6/ 1	6/ 1
ChannelRouting	11	6/ 2	**6/2**	**6/2**	**6/2**	**6/2**	5/ 2	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**
Solitaire	27	19/ 0	2/ 0	5/ 0	1/ 0	4/ 0	0/ 0	0/ 0	0/ 0	0/ 0	**21/0**	**21/0**	20/ 0	**21/0**
Labyrinth	29	26/ 0	**1/0**	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0
WeightBoundedDominatingSet	29	26/ 0	18/ 0	18/ 0	17/ 0	18/ 0	12/ 0	12/ 0	11/ 0	12/ 0	**22/0**	**22/0**	**22/0**	21/ 0
MazeGeneration	29	10/15	8/15	1/15	0/15	0/16	5/16	1/15	0/15	1/15	**10/17**	10/15	5/15	4/15
15Puzzle	16	16/ 0	**16/0**	15/ 0	14/ 0	15/ 0	4/ 0	4/ 0	5/ 0	5/ 0	0/ 0	0/ 0	0/ 0	0/ 0
BlockedNQueens	29	15/14	2/ 2	0/ 2	1/ 2	0/ 2	1/ 0	2/ 0	2/ 0	0/ 0	**15/13**	**15/13**	15/12	**15/13**
ConnectedDominatingSet	21	10/10	**10/11**	9/ 8	**10/11**	6/ 3	10/10	9/10	10/ 9	10/ 9	9/ 8	7/ 6	9/ 7	7/ 6
EdgeMatching	29	29/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	**3/0**	1/ 0	**3/0**	2/ 0
Fastfood	29	10/19	9/16	10/16	10/16	9/16	9/ 9	9/ 9	9/10	9/ 9	**10/18**	**10/18**	**10/18**	**10/18**
GeneralizedSlitherlink	29	29/ 0	**29/0**	20/ 0	**29/0**	**29/0**	**29/0**	**29/0**	16/ 0	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**
HamiltonianPath	29	29/ 0	27/ 0	25/ 0	**29/0**	28/ 0	26/ 0	27/ 0	25/ 0	26/ 0	**29/0**	**29/0**	**29/0**	**29/0**
Hanoi	15	15/ 0	**15/0**	**15/0**	**15/0**	**15/0**	5/ 0	5/ 0	5/ 0	4/ 0	**15/0**	**15/0**	**15/0**	**15/0**
HierarchicalClustering	12	8/ 4	**8/4**	**8/4**	**8/4**	**8/4**	4/ 4	4/ 4	4/ 4	4/ 4	**8/4**	**8/4**	**8/4**	**8/4**
SchurNumbers	29	13/16	6/16	5/16	5/16	5/16	9/16	9/16	9/16	9/16	**11/16**	**11/16**	**11/16**	**11/16**
Sokoban	29	9/20	9/19	8/19	8/19	8/19	7/15	7/13	7/14	5/13	**9/20**	**9/20**	**9/20**	**9/20**
Sudoku	10	10/ 0	5/ 0	4/ 0	4/ 0	5/ 0	4/ 0	4/ 0	4/ 0	4/ 0	**9/0**	8/ 0	8/ 0	**9/0**
TravellingSalesperson	29	29/ 0	3/ 0	0/ 0	6/ 0	10/ 0	0/ 0	8/ 0	0/ 0	0/ 0	**29/0**	**29/0**	7/ 0	7/ 0

Table 1 collects the results from our experiments without normalization whereas Table 2 shows the results when LP2NORMAL [19] was used to remove extended rule types discussed in Section 4. In both tables, the first column gives the name of the benchmark, followed by the number of instances of that particular benchmark in the second column. The following columns indicate the numbers of instances that were solved by the systems considered in our experiments. A notation like 8/4 means that the system was able to solve eight satisfiable and four unsatisfiable instances in that particular benchmark. Hence, if there are 15 instances in a benchmark and the system could only solve 8/4, this means that the system was unable to solve the remaining three instances within the time limit of 600 seconds, i.e. ten minutes, per instance[15]. As regards the number of solved instances in each benchmark, the best performing translation-based approaches are highlighted in boldface. Though it was not shown in all tables, we also run the experiments using translator LP2DIFF with Z3 as back-end solver, and the summary is included in Table 3—giving an overview of experimental results in terms of total numbers of instances solved out of 516.

It is apparent that the systems based on LP2BV did not perform very well without normalization. As indicated by Table 3, the overall performance was even worse than that of systems using LP2DIFF for translation and Z3 for model search. However, if the input was first translated into a normal logic program using LP2NORMAL, i.e., before translation into a bit-vector theory, the performance was clearly better. Actually, it exceeded that of the systems based on LP2DIFF and became closer to that of CLASP. We note that normalization does not help so much in case of LP2DIFF and the experimental results obtained using both normalized and unnormalized instances are quite similar in terms of solved instances. Thus it seems that solvers for bit-vector logic are not able to make the best of native translations of cardinality and weight rules from Section 4 in full. If an analogous translation into difference logic is used, as implemented in LP2DIFF, such a negative effect was not perceived using Z3. Our understanding is that the efficient graph-theoretic satisfiability check for difference constraints used in the search procedure of Z3 turns the native translation feasible as well. As indicated by our test results, BOOLECTOR is clearly better back-end solver for LP2BV than Z3. This was to be expected since BOOLECTOR is a native solver for bit-vector

[15] One observation is that the performance of systems based on LP2BV is quite stable: even when we extended the time limit to 20 minutes, the results did not change much (differences of only one or two instances were perceived in most cases).

Table 2. Experimental results with normalization

Benchmark	INST	CLASP	LP2BV+BOOLECTOR				LP2BV+Z3			
			W	L	G	LG	W	L	G	LG
Overall Performance	516	459	**381**	343	379	**381**	346	330	325	331
		346/113	**279/102**	243/100	278/101	**281/100**	240/106	231/ 99	224/101	232/ 99
KnightTour	10	10/ 0	**2/0**	**2/0**	1/ 0	0/ 0	1/ 0	0/ 0	0/ 0	0/ 0
GraphColouring	29	9/ 0	8/ 0	8/ 0	8/ 0	8/ 0	**9/2**	**9/2**	**9/2**	**9/2**
WireRouting	23	11/11	2/ 6	1/ 3	1/ 3	1/ 3	**2/7**	1/ 4	1/ 4	1/ 3
DisjunctiveScheduling	10	5/ 0	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**
GraphPartitioning	13	4/ 1	**5/0**	**5/0**	4/ 0	**5/0**	2/ 1	2/ 1	2/ 1	2/ 0
ChannelRouting	11	6/ 2	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**
Solitaire	27	18/ 0	**23/0**	**23/0**	**23/0**	**23/0**	22/ 0	22/ 0	22/ 0	22/ 0
Labyrinth	29	27/ 0	1/ 0	1/ 0	2/ 0	**3/0**	0/ 0	0/ 0	0/ 0	0/ 0
WeightBoundedDominatingSet	29	25/ 0	15/ 0	15/ 0	15/ 0	**16/0**	10/ 0	10/ 0	10/ 0	10/ 0
MazeGeneration	29	10/15	**8/15**	0/15	0/15	0/16	5/16	0/15	0/15	0/15
15Puzzle	16	15/ 0	**16/0**	**16/0**	**16/0**	**16/0**	11/ 0	10/ 0	11/ 0	11/ 0
BlockedNQueens	29	15/14	14/14	14/14	14/14	14/14	**15/14**	**15/14**	**15/14**	**15/14**
ConnectedDominatingSet	21	10/11	**10/11**	8/11	9/11	9/10	**10/11**	9/11	9/11	9/11
EdgeMatching	29	29/ 0	29/ 0	29/ 0	29/ 0	29/ 0	29/ 0	29/ 0	29/ 0	29/ 0
Fastfood	29	10/19	9/14	9/15	**9/16**	9/15	0/13	0/10	0/12	0/12
GeneralizedSlitherlink	29	29/ 0	**29/ 0**	21/ 0	**29/ 0**	**29/ 0**	**29/ 0**	**29/ 0**	21/ 0	**29/ 0**
HamiltonianPath	29	29/ 0	**29/ 0**	28/ 0	**29/ 0**	**29/ 0**	**29/ 0**	**29/ 0**	**29/ 0**	**29/ 0**
Hanoi	15	15/ 0	**15/ 0**	**15/ 0**	**15/ 0**	**15/ 0**	**15/ 0**	**15/ 0**	**15/ 0**	**15/ 0**
HierarchicalClustering	12	8/ 4	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**
SchurNumbers	29	13/16	10/16	10/16	9/16	10/16	**13/16**	**13/16**	**13/16**	**13/16**
Sokoban	29	9/20	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**
Sudoku	10	10/ 0	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**
TravellingSalesperson	29	29/ 0	16/ 0	0/ 0	**27/0**	**27/0**	0/ 0	0/ 0	0/ 0	0/ 0

logic whereas Z3 supports a wider variety of SMT fragments and can be used for more general purposes. Moreover, the design of LP2BV takes into account operators of bit-vector logic which are directly supported by BOOLECTOR and not implemented as syntactic sugar.

In addition, we note on the basis of our results that the performance of the state-of-the-art ASP solver CLASP is significantly better, and the translation-based approaches to computing stable models are still left behind. By the results of Table 2, even the best variants of systems based on LP2BV did not work well enough to compete with CLASP. The difference is especially due to the following benchmarks: *Knight Tour*, *Wire Routing*, *Graph Partitioning*, *Labyrinth*, *Weight Bounded Dominating Set*, *Fastfood*, and *Travelling Salesperson*. All of them involve either recursive rules (*Knight Tour*, *Wire Routing*, and *Labyrinth*), weight rules (*Weight Bounded Dominating Set* and *Fastfood*), or both (*Graph Partitioning* and *Travelling Salesperson*). Hence, it seems that handling recursive rules and weight constraints in the translational approach is less efficient compared to their native implementation in CLASP. When using the current normalization techniques to remove cardinality and weight rules, the sizes of ground programs tend to increase significantly and, in particular, if weight rules are abundant. For example, after normalization the ground programs are ten times larger for the benchmark *Weight Bounded Dominating Set*, and five times larger for *Fastfood*. It is also worth pointing out that the efficiency of CLASP turned out to be insensitive to normalization.

While having trouble with recursive rules and weight constraints for particular benchmarks, the translational approach handles certain large instances quite well. The largest instances in the experiments belong to the *Disjunctive Scheduling* benchmark, of which all instances are ground programs of size over one megabyte but after normalization[16], the LP2BV systems can solve as many instances as CLASP.

6 Conclusion

In this paper, we present a novel and concise translation from normal logic programs into fixed-width bit-vector theories. Moreover, the extended rule types supported by SMODELS compatible answer set solvers can be covered via native translations. The length of the resulting translation is linear with respect to the

[16] In this benchmark, normalization does not affect the size of grounded programs significantly.

Table 3. Summary of the experimental results

System	W	L	G	LG
LP2BV+BOOLECTOR	276	244	261	256
LP2BV+Z3	217	216	194	204
LP2DIFF+Z3	360	349	324	324
CLASP	465			
LP2NORMAL2BV+BOOLECTOR	381	343	379	381
LP2NORMAL2BV+Z3	346	330	325	331
LP2NORMAL2DIFF+Z3	364	357	349	349
LP2NORMAL+CLASP	459			

length of the original program. The translation has been implemented as a translator, LP2BV, which enables the use of bit-vector solvers in the search for answer sets. Our preliminary experimental results indicate a level of performance which is similar to that obtained using solvers for difference logic. However, this presumes one first to translate extended rule types into normal rules and then to apply the translation into bit-vector logic. One potential explanation for such behavior is the way in which SMT solvers implement reasoning with bit vectors: a predominant strategy is to translate theory atoms involving bit vectors into propositional formulas and to apply satisfiability checking techniques systematically. We anticipate that an improved performance could be obtained if a native support for certain bit vector primitives were incorporated into SMT solvers directly. When comparing to the state-of-the-art ASP solver CLASP, we noticed that the performance of the translation based approach compared unfavorably, in particular, for benchmarks which contained recursive rules or weight constraints or both. This indicates that the performance can be improved by developing new translation techniques for these two features. In order to obtain a more comprehensive view of the performance characteristics of the translational approach, the plan is to extend our experimental setup to include benchmarks that were used in the third ASP competition [7]. Moreover, we intend to use the new SMT library format [4] in future versions of our translators.

Acknowledgments This research has been partially funded by the Academy of Finland under the project "*Methods for Constructing and Solving Large Constraint Models*" (MCM, #122399).

References

1. Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.
2. Marcello Balduccini. Industrial-size scheduling with ASP+CP. In Delgrande and Faber [10], pages 284–296.
3. Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [5], pages 825–885.
4. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard version 2.0.
5. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
6. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bitvectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.
7. Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In Delgrande and Faber [10], pages 388–403.
8. Keith Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.
9. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

10. James Delgrande and Wolfgang Faber, editors. *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*. Springer, 2011.
11. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In Erdem et al. [12], pages 637–654.
12. Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer, 2009.
13. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.
14. Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In Patricia Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2009.
15. Michael Gelfond and Nicola Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.
16. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.
17. Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1–2):35–86, June 2006.
18. Tomi Janhunen, Guohua Liu, and Ilkka Niemelä. Tight integration of non-ground answer set programming and satisfiability modulo theories. In Eugenia Ternovska and David Mitchell, editors, *Working Notes of Grounding and Transformations for Theories with Variables*, pages 1–13, Vancouver, Canada, May 2011.
19. Tomi Janhunen and Ilkka Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2011.
20. Tomi Janhunen, Ilkka Niemelä, and Mark Sevalnev. Computing stable models via reductions to difference logic. In Erdem et al. [12], pages 142–154.
21. Victor Marek and Venkatramana Subrahmanian. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theor. Comput. Sci.*, 103(2):365–386, 1992.
22. Victor Marek and Mirek Truszczyński. Stable models and an alternative logicprogramming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.
23. Veena Mellarkod and Michael Gelfond. Integrating answer set reasoning with constraint solving techniques. In Jacques Garrigue and Manuel Hermenegildo, editors, *FLOPS*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer, 2008.
24. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.
25. Ilkka Niemelä. Stable models and difference logic. *Ann. Math. Artif. Intell.*, 53(1-4):313–329, 2008.
26. Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In Kousha Etessami and Sriram Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.
27. Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal.
28. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

Making Use of Advances in Answer-Set Programming for Abstract Argumentation Systems*

Wolfgang Dvořák, Sarah Alice Gaggl, Johannes Wallner, and Stefan Woltran

Institute of Information Systems, Database and Artificial Intelligence Group,
Vienna University of Technology, Favoritenstrae 9-11, 1040 Wien, Austria
EMail: {dvorak, gaggl, wallner, woltran}@dbai.tuwien.ac.at

Abstract. Dung's famous abstract argumentation frameworks represent the core formalism for many problems and applications in the field of argumentation which significantly evolved within the last decade. Recent work in the field has thus focused on implementations for these frameworks, whereby one of the main approaches is to use Answer-Set Programming (ASP). While some of the argumentation semantics can be nicely expressed within the ASP language, others required rather cumbersome encoding techniques. Recent advances in ASP systems, in particular, the `metasp` optimization front-end for the ASP-package `gringo/claspD` provides direct commands to filter answer sets satisfying certain subset-minimality (or -maximality) constraints. This allows for much simpler encodings compared to the ones in standard ASP language. In this paper, we experimentally compare the original encodings (for the argumentation semantics based on preferred, semi-stable, and respectively, stage extensions) with new `metasp` encodings. Moreover, we provide novel encodings for the recently introduced resolution-based grounded semantics. Our experimental results indicate that the `metasp` approach works well in those cases where the complexity of the encoded problem is adequately mirrored within the `metasp` approach.

Keywords: Abstract Argumentation, Answer-Set Programming, Metasp

1 Introduction

In Artificial Intelligence (AI), the area of argumentation (the survey by Bench-Capon and Dunne [3] gives an excellent overview) has become one of the central issues during the last decade. Although there are now several branches within this area, there is a certain agreement that Dung's famous abstract argumentation frameworks (AFs) [7] still represent the core formalism for many of the problems and applications in the field. In a nutshell, AFs formalize statements together with a relation denoting rebuttals between them, such that the semantics gives a handle to solve the inherent conflicts between statements by selecting admissible subsets of them, but without taking the concrete contents of the statements into account. Several semantical principles how to select those subsets have already been proposed by Dung [7] but numerous other proposals have been made over the last years. In this paper we shall focus on the preferred [7], semi-stable [4], stage [17], and the resolution-based grounded semantics [1]. Each of these semantics is based on some kind of $\subseteq$-maximality (resp. -minimality) and thus is well amenable for the novel `metasp` concepts which we describe below.

Let us first talk about the general context of the paper, which is the realization of abstract argumentation within the paradigm of Answer-Set Programming (see [16] for an overview). We follow here the ASPARTIX[1] approach [11], where a single program is used to encode a particular argumentation semantics, while the instance of an argumentation framework is given as an input database. For problems located on the second level of the polynomial hierarchy (i.e. for preferred, stage, and semi-stable semantics) ASP encodings turned out to be quite complicated and hardly accessible for non-experts in ASP (we will sketch here the encoding for the stage semantics in some detail, since it has not been presented in [11]). This is due to the fact that tests for subset-maximality have to be done "by hand" in ASP requiring a certain saturation technique. However, recent advances in ASP solvers, in particular, the `metasp` optimization

* Supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-028.

[1] See http://rull.dbai.tuwien.ac.at:8080/ASPARTIX for a web front-end of ASPARTIX.

front-end for the ASP-system `gringo/claspD` allows for much simpler encodings for such tests. More precisely, `metasp` allows to use the traditional $\#minimize$ statement (which in its standard variant minimizes wrt. cardinality or weights, but not wrt. subset inclusion) also for selection among answer sets which are minimal (or maximal) wrt. subset inclusion in certain predicates. Details about `metasp` can be found in [13].

Our first main contribution will be the practical comparison between handcrafted encodings (i.e. encodings in the standard ASP language without the new semantics for the $\#minimize$ statement) and the much simpler `metasp` encodings for argumentation semantics. The experiments show that the `metasp` encodings do not necessarily result in longer runtimes. In fact, the `metasp` encodings for the semantics located on the second level of the polynomial hierarchy outperform the handcrafted saturation-based encodings. We thus can give additional evidence to the observations in [13], where such a speed-up was reported for encodings in a completely different application area.

Our second contribution is the presentation of ASP encodings for the resolution-based grounded semantics [1]. To the best of our knowledge, no implementation for this quite interesting semantics has been released so far. In this paper, we present a rather involved handcrafted encoding (basically following the NP-algorithm presented in [1]) but also two much simpler encodings (using `metasp`) which rely on the original definition of the semantics.

Our results indicate that `metasp` is a very useful tool for problems known to be hard for the second-level, but one might loose performance in case `metasp` is used for "easier" problems just for the sake of comfortability. Nonetheless, we believe that the concept of the advanced $\#minimize$ statement is vital for ASP, since it allows for rapid prototyping of second-level encodings without being an ASP guru.

The remainder of the paper is organized as follows: Section 2 provides the necessary background. Section 3 then contains the ASP encodings for the semantics we are interested in here. We first discuss the handcrafted saturation-based encoding for stage semantics (the ones for preferred and semi-stable are similar and already published). Then, in Section 3.2 we provide the novel `metasp` encodings for all considered semantics. Afterwards, in Section 3.3 we finally present an alternative encoding for the resolution-based grounded semantics which better mirrors the complexity of this semantics. Section 4 then presents our experimental evaluation. We conclude the paper with a brief summary and discussion for future research directions.

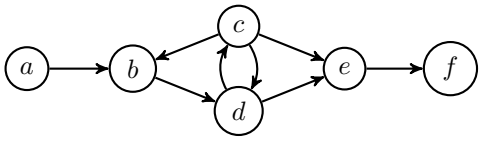
2 Background

2.1 Abstract Argumentation

In this section we introduce (abstract) argumentation frameworks [7] and recall the semantics we study in this paper (see also [1, 2]). Moreover, we highlight complexity results for typical decision problems associated to such frameworks.

Definition 1. *An* argumentation framework (AF) *is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation. The pair $(a, b) \in R$ means that a attacks b. An argument $a \in A$ is* defended *by a set $S \subseteq A$ if, for each $b \in A$ such that $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$.*

Example 1. Consider the AF $F = (A, R)$ with $A = \{a, b, c, d, e, f\}$ and $R = \{(a, b), (b, d), (c, b), (c, d), (c, e), (d, c), (d, e), (e, f)\}$, and the graph representation of F:



Semantics for argumentation frameworks are given via a function σ which assigns to each AF $F = (A, R)$ a set $\sigma(F) \subseteq 2^A$ of extensions. We shall consider here for σ the functions stb, adm, com, prf, grd, grd^*, stg, and sem which stand for stable, admissible, complete, preferred, grounded, resolution-based grounded, stage, and semi-stable semantics respectively. Towards the definition of these semantics we have to introduce two more formal concepts.

Definition 2. *Given an AF $F = (A, R)$. The characteristic function $\mathcal{F}_F : 2^A \Rightarrow 2^A$ of F is defined as $\mathcal{F}_F(S) = \{x \in A \mid x \text{ is defended by } S\}$. Moreover, for a set $S \subseteq A$, we denote the set of arguments attacked by S as $S_R^{\oplus} = \{x \mid \exists y \in S \text{ such that } (y, x) \in R\}$, and define the* range of S as $S_R^+ = S \cup S_R^{\oplus}$.

Definition 3. *Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is* conflict-free (in F)*, if there are no $a, b \in S$, such that $(a, b) \in R$. $cf(F)$ denotes the collection of conflict-free sets of F. For a conflict-free set $S \in cf(F)$, it holds that*

- *$S \in stb(F)$, if $S_R^+ = A$;*
- *$S \in adm(F)$, if $S \subseteq \mathcal{F}_F(S)$;*
- *$S \in com(F)$, if $S = \mathcal{F}_F(S)$;*
- *$S \in grd(F)$, if $S \in com(F)$ and there is no $T \in com(F)$ with $T \subset S$;*
- *$S \in prf(F)$, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T \supset S$;*
- *$S \in sem(F)$, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T_R^+ \supset S_R^+$;*
- *$S \in stg(F)$, if there is no $T \in cf(F)$ in F, such that $T_R^+ \supset S_R^+$.*

We recall that for each AF F, the grounded semantics yields a unique extension, the grounded extension, which is the least fix-point of the characteristic function $\mathcal{F}_F$.

Example 2. Consider the AF F from Example 1. We have $\{a, d, f\}$ and $\{a, c, f\}$ as the stable extensions and thus $stb(F) = stg(F) = sem(F) = \{\{a, d, f\}, \{a, c, f\}\}$. The admissible sets of F are $\{\}$, $\{a\}$, $\{c\}$, $\{a, c\}$, $\{a, d\}$, $\{c, f\}$, $\{a, c, f\}$, $\{a, d, f\}$ and therefore $prf(F) = \{\{a, c, f\}, \{a, d, f\}\}$. Finally we have $com(F) = \{\{a\}, \{a, c, f\}, \{a, d, f\}\}$, with $\{a\}$ being the grounded extension.

On the base of these semantics one can define the family of resolution-based semantics [1], with the resolution-based grounded semantics being the most popular instance.

Definition 4. *A resolution $\beta \subset R$ of an $F = (A, R)$ contains exactly one of the attacks (a, b), (b, a) if $\{(a, b), (b, a)\} \subseteq R$, $a \neq b$, and no further attacks. A set $S \subseteq A$ is a resolution-based grounded extension of F if (i) there exists a resolution β such that $S = grd((A, R \setminus \beta))$;[2] and (ii) there is no resolution β' such that $grd((A, R \setminus \beta')) \subset S$.*

Example 3. Recall the AF $F = (A, F)$ from Example 1. There is one mutual attack and thus we have two resolutions $\beta_1 = \{(c, d)\}$ and $\beta_2 = \{(d, c)\}$. Definition 4 gives us two candidates, namely $grd((A, R \setminus \beta_1)) = \{a, d, f\}$ and $grd((A, R \setminus \beta_2)) = \{a, c, f\}$; as they are not in $\subset$-relation they are the resolution-based grounded extensions of F.

We now turn to the complexity of reasoning in AFs. To this end, we define the following decision problems for the semantics σ introduced in Definitions 3 and 4:

- *Credulous Acceptance* Cred_σ: Given AF $F = (A, R)$ and an argument $a \in A$. Is a contained in some $S \in \sigma(F)$?
- *Skeptical Acceptance* Skept_σ: Given AF $F = (A, R)$ and an argument $a \in A$. Is a contained in each $S \in \sigma(F)$?
- *Verification of an extension* Ver_σ: Given AF $F = (A, R)$ and a set of arguments $S \subseteq A$. Is $S \in \sigma(F)$?

We assume the reader has knowledge about standard complexity classes like P and NP and recall that Σ_2^P is the class of decision problems that can be decided in polynomial time using a nondeterministic Turing machine with access to an NP-oracle. The class Π_2^P is defined as the complementary class of Σ_2^P, i.e. $\Pi_2^P = \text{co}\Sigma_2^P$.

In Table 1 we summarize complexity results relevant for our work [1, 6, 8–10].

[2] Abusing notation slightly, we use $grd(F)$ for denoting the unique grounded extension of F.

	prf	*sem*	*stg*	*grd**
Cred_σ	NP-c	Σ_2^P-c	Σ_2^P-c	NP-c
Skept_σ	Π_2^P-c	Π_2^P-c	Π_2^P-c	coNP-c
Ver_σ	coNP-c	coNP-c	coNP-c	in P

Table 1. Complexity of abstract argumentation ($\mathcal{C}$-c denotes completeness for class $\mathcal{C}$)

2.2 Answer-Set Programming

We first give a brief overview of the syntax and semantics of disjunctive logic programs under the answer-sets semantics [14]; for further background, see [15].

We fix a countable set $\mathcal{U}$ of *(domain) elements*, also called *constants*; and suppose a total order $<$ over the domain elements. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity $n \geq 0$ and each t_i is either a variable or an element from $\mathcal{U}$. An atom is *ground* if it is free of variables. $B_\mathcal{U}$ denotes the set of all ground atoms over $\mathcal{U}$.

A *(disjunctive) rule* r is of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \; not\, b_{k+1}, \ldots, \; not\, b_m,$$

with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, and "not" stands for *default negation*. The *head* of r is the set $H(r) = \{a_1, \ldots, a_n\}$ and the *body* of r is $B(r) = \{b_1, \ldots, b_k, not\, b_{k+1}, \ldots, \; not\, b_m\}$. Furthermore, $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule r is *normal* if $n \leq 1$ and a *constraint* if $n = 0$. A rule r is *safe* if each variable in r occurs in $B^+(r)$. A rule r is *ground* if no variable occurs in r. A *fact* is a ground rule without disjunction and empty body. An *(input) database* is a set of facts. A program is a finite set of disjunctive rules. For a program π and an input database D, we often write $\pi(D)$ instead of $D \cup \pi$. If each rule in a program is normal (resp. ground), we call the program normal (resp. ground). Besides disjunctive and normal program, we consider here the class of optimization programs, i.e. normal programs which additionally contain $\#minimize$ statements

$$\#minimize[l_1 = w_1@J_1, \ldots, l_k = w_k@J_k], \tag{1}$$

where l_i is a literal, w_i an integer weight and J_i an integer priority level.

For any program π, let U_π be the set of all constants appearing in π. $Gr(\pi)$ is the set of rules $r\sigma$ obtained by applying, to each rule $r \in \pi$, all possible substitutions σ from the variables in r to elements of U_π. An *interpretation* $I \subseteq B_\mathcal{U}$ *satisfies* a ground rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program π, if each $r \in \pi$ is satisfied by I. A non-ground rule r (resp., a program π) is satisfied by an interpretation I iff I satisfies all groundings of r (resp., $Gr(\pi)$). $I \subseteq B_\mathcal{U}$ is an *answer set* of π iff it is a subset-minimal set satisfying the *Gelfond-Lifschitz reduct* $\pi^I = \{H(r) \leftarrow B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Gr(\pi)\}$. For a program π, we denote the set of its answer sets by $\mathcal{AS}(\pi)$.

For semantics of optimization programs, we interpret the $\#minimize$ statement wrt. subset-inclusion: For any sets X and Y of atoms, we have $Y \subseteq^w_J X$, if for any weighted literal $l = w@J$ occurring in (1), $Y \models l$ implies $X \models l$. Then, M is a collection of relations of the form $\subseteq^w_J$ for priority levels J and weights w. A standard answer set (i.e. not taking the minimize statements into account) Y of π *dominates* a standard answer set X of π wrt. M if there are a priority level J and a weight w such that $X \subseteq^w_J Y$ does not hold for $\subseteq^w_J \in M$, while $Y \subseteq^{w'}_{J'} X$ holds for all $\subseteq^{w'}_{J'} \in M$ where $J' \geq J$. Finally a standard answer set X is an answer set of an optimization program π wrt. M if there is no standard answer set Y of π that dominates X wrt. M.

Credulous and skeptical reasoning in terms of programs is defined as follows. Given a program π and a set of ground atoms A. Then, we write $\pi \models_c A$ (credulous reasoning), if A is contained in some answer set of π; we write $\pi \models_s A$ (skeptical reasoning), if A is contained in each answer set of π.

We briefly recall some complexity results for disjunctive logic programs. In fact, since we will deal with fixed programs we focus on results for data complexity. Depending on the concrete definition of $\models$,

e	normal programs	disjunctive program	optimization programs
$\models_c$	NP	Σ_2^P	Σ_2^P
$\models_s$	coNP	Π_2^P	Π_2^P

Table 2. Data Complexity for logic programs (all results are completeness results).

we give the complexity results in Table 2 (cf. [5] and the references therein). We note here, that even normal programs together with the optimization technique have a worst case complexity of Σ_2^P (resp. Π_2^P). Inspecting Table 1 one can see which kind of encoding is appropriate for an argumentation semantics.

3 Encodings of AF Semantics

In this section we first show how to represent AFs in ASP and we discuss three programs which we need later on in this section[3]. Then, in Subsection 3.1 we exemplify on the stage semantics the saturation technique for encodings which solve associated problems which are on the second level of the polynomial hierarchy. In Subsection 3.2 we will make use of the newly developed `metasp` optimization technique. In Subsection 3.3 we give an alternative encoding based on the algorithm of Baroni *et al.* in [1], which respects the lower complexity of resolution-based grounded semantics.

All our programs are fixed which means that the only translation required, is to give an AF F as input database $\hat{F}$ to the program π_σ for a semantics σ. In fact, for an AF $F = (A, R)$, we define $\hat{F}$ as

$$\hat{F} = \{\ \mathrm{arg}(a) \mid a \in A\} \cup \{\mathrm{defeat}(a, b) \mid (a, b) \in R\ \}.$$

In what follows, we use unary predicates $\mathrm{in}/1$ and $\mathrm{out}/1$ to perform a guess for a set $S \subseteq A$, where $\mathrm{in}(a)$ represents that $a \in S$. The following notion of correspondence is relevant for our purposes.

Definition 5. *Let $\mathcal{S} \subseteq 2^{\mathcal{U}}$ be a collection of sets of domain elements and let $\mathcal{I} \subseteq 2^{B_\mathcal{U}}$ be a collection of sets of ground atoms. We say that $\mathcal{S}$ and $\mathcal{I}$ correspond to each other, in symbols $\mathcal{S} \cong \mathcal{I}$, iff (i) for each $S \in \mathcal{S}$, there exists an $I \in \mathcal{I}$, such that $\{a \mid \mathrm{in}(a) \in I\} = S$; (ii) for each $I \in \mathcal{I}$, it holds that $\{a \mid \mathrm{in}(a) \in I\} \in \mathcal{S}$; and (iii) $|\mathcal{S}| = |\mathcal{I}|$.*

Consider an AF F. The following program fragment guesses, when augmented by $\hat{F}$, any subset $S \subseteq A$ and then checks whether the guess is conflict-free in F:

$$\begin{aligned}\pi_{cf} = \{\ & \mathrm{in}(X) \leftarrow \mathit{not}\ \mathrm{out}(X), \mathrm{arg}(X);\\ & \mathrm{out}(X) \leftarrow \mathit{not}\ \mathrm{in}(X), \mathrm{arg}(X);\\ & \leftarrow \mathrm{in}(X), \mathrm{in}(Y), \mathrm{defeat}(X, Y)\ \}.\end{aligned}$$

Proposition 1. *For any AF F, $cf(F) \cong \mathcal{AS}(\pi_{cf}(\hat{F}))$.*

Sometimes we have to avoid the use of negation. This might either be the case for the saturation technique or if a simple program can be solved without a Guess&Check approach. Then, encodings typically rely on a form of loops where all domain elements are visited and it is checked whether a desired property holds for all elements visited so far. We will use this technique in our saturation-based encoding in the upcoming subsection, but also for computing the grounded extension in Subsection 3.2. For this purpose the program $\pi_<$, which is taken from [11], is used to encode the infimum, successor and supremum of an order $<$ over the domain elements in the predicates $\mathrm{inf}/1, \mathrm{succ}/2$ and $\mathrm{sup}/1$ respectively. The order over the domain elements is usually provided by common ASP solvers.

Finally, the following module computes for a guessed subset $S \subseteq A$ the range S_R^+ (see Def. 2) of S in an AF (A, R).

[3] We make use of some program modules already defined in [11].

$$\begin{aligned}\pi_{range} = \{&\mathrm{in\_range}(X) \leftarrow \mathrm{in}(X);\\ &\mathrm{in\_range}(X) \leftarrow \mathrm{in}(Y), \mathrm{defeat}(Y,X);\\ &\mathrm{not\_in\_range}(X) \leftarrow \mathrm{arg}(X), \mathit{not}\, \mathrm{in\_range}(X)\}.\end{aligned}$$

3.1 Saturation Encodings

In this subsection we make use of the saturation technique introduced by Eiter and Gottlob in [12]. In [11], this technique was already used to encode the preferred and semi-stable semantics. Here we give the encodings for the stage semantics, which is similar to the one of semi-stable semantics, to exemplify the use of the saturation technique.

In fact, for an AF $F = (A, R)$ and $S \in cf(F)$ we need to check whether no $T \in cf(F)$ with $S_R^+ \subset T_R^+$ exists. Therefore we have to guess an arbitrary set T and saturate in case (i) T is not conflict-free, and (ii) $S_R^+ \not\subset T_R^+$. Together with π_{cf} this is done with the following module, where $\mathrm{in}/1$ holds the current guess for S and $\mathrm{inN}/1$ holds the current guess for T. More specifically, rule $\mathrm{fail} \leftarrow \mathrm{inN}(X), \mathrm{inN}(Y), \mathrm{defeat}(X,Y)$ checks for (i) and the remaining two rules with fail in the head fire in case $S_R^+ = T_R^+$ (indicated by predicate $\mathrm{eqplus}/0$ described below), or there exists an $a \in S_R^+$ such that $a \notin T_R^+$ (here we use predicate $\mathrm{in\_range}/1$ from above and predicate $\mathrm{not\_in\_rangeN}/1$ which we also present below). As is easily checked one of these two conditions holds exactly if (ii) holds.

$$\begin{aligned}\pi_{satstage} = \{\ &\mathrm{inN}(X) \vee \mathrm{outN}(X) \leftarrow \mathrm{arg}(X);\\ &\mathrm{fail} \leftarrow \mathrm{inN}(X), \mathrm{inN}(Y), \mathrm{defeat}(X,Y);\\ &\mathrm{fail} \leftarrow \mathrm{eqplus};\\ &\mathrm{fail} \leftarrow \mathrm{in\_range}(X), \mathrm{not\_in\_rangeN}(X);\\ &\mathrm{inN}(X) \leftarrow \mathrm{fail}, \mathrm{arg}(X);\\ &\mathrm{outN}(X) \leftarrow \mathrm{fail}, \mathrm{arg}(X);\\ &\leftarrow \mathit{not}\, \mathrm{fail}\ \}.\end{aligned}$$

For the definition of predicates $\mathrm{not\_in\_rangeN}/1$ and $\mathrm{eqplus}/0$ we make use of the aforementioned loop technique and predicates from program $\pi_<$.

$$\begin{aligned}\pi_{rangeN} = \{\ &\mathrm{undefeated\_upto}(X,Y) \leftarrow \mathrm{inf}(Y), \mathrm{outN}(X), \mathrm{outN}(Y);\\ &\mathrm{undefeated\_upto}(X,Y) \leftarrow \mathrm{inf}(Y), \mathrm{outN}(X), \mathit{not}\, \mathrm{defeat}(Y,X);\\ &\mathrm{undefeated\_upto}(X,Y) \leftarrow \mathrm{succ}(Z,Y), \mathrm{undefeated\_upto}(X,Z), \mathrm{outN}(Y);\\ &\mathrm{undefeated\_upto}(X,Y) \leftarrow \mathrm{succ}(Z,Y), \mathrm{undefeated\_upto}(X,Z),\\ &\qquad\qquad \mathit{not}\, \mathrm{defeat}(Y,X);\\ &\mathrm{not\_in\_rangeN}(X) \leftarrow \mathrm{sup}(Y), \mathrm{outN}(X), \mathrm{undefeated\_upto}(X,Y);\\ &\mathrm{in\_rangeN}(X) \leftarrow \mathrm{inN}(X);\\ &\mathrm{in\_rangeN}(X) \leftarrow \mathrm{outN}(X), \mathrm{inN}(Y), \mathrm{defeat}(Y,X)\ \}.\end{aligned}$$

$$\begin{aligned}\pi_{eq}^+ = \{\ &\mathrm{eqp\_upto}(X) \leftarrow \mathrm{inf}(X), \mathrm{in\_range}(X), \mathrm{in\_rangeN}(X);\\ &\mathrm{eqp\_upto}(X) \leftarrow \mathrm{inf}(X), \mathrm{not\_in\_range}(X), \mathrm{not\_in\_rangeN}(X);\\ &\mathrm{eqp\_upto}(X) \leftarrow \mathrm{succ}(Z,X), \mathrm{in\_range}(X), \mathrm{in\_rangeN}(X), \mathrm{eqp\_upto}(Z);\\ &\mathrm{eqp\_upto}(X) \leftarrow \mathrm{succ}(Y,X), \mathrm{not\_in\_range}(X), \mathrm{not\_in\_rangeN}(X), \mathrm{eqp\_upto}(Y);\\ &\mathrm{eqplus} \leftarrow \mathrm{sup}(X), \mathrm{eqp\_upto}(X)\ \};\end{aligned}$$

Proposition 2. *For any AF F, $stg(F) \cong \mathcal{AS}(\pi_{stg}(\hat{F}))$, where $\pi_{stg} = \pi_{cf} \cup \pi_{<} \cup \pi_{range} \cup \pi_{rangeN} \cup \pi^{+}_{eq} \cup \pi_{satstage}$.*

3.2 Meta ASP Encodings

The following encodings for preferred, semi-stable and stage semantics are written using the $\#minimize[\cdot]$ statement when evaluated with the subset minimization semantics provided by `metasp`. For our encodings we do not need prioritization and weights, therefore these are omitted (i.e. set to default) in the minimization statements. The fact `optimize(1,1,incl)` is added to the meta ASP encodings, to indicate that we use subset inclusion for the optimization technique using priority and weight 1.

We now look at the encodings for the preferred, semi-stable and stage semantics using this minimization technique. First we need one auxiliary module for admissible extensions.

$$\begin{aligned}\pi_{adm} = \pi_{cf} \cup \{&\mathrm{defeated}(X) \leftarrow \mathrm{in}(Y), \mathrm{defeat}(Y,X);\\ &\leftarrow \mathrm{in}(X), \mathrm{defeat}(Y,X), \mathit{not}\ \mathrm{defeated}(Y)\}.\end{aligned}$$

Now the modules for preferred, semi-stable and stage semantics are easy to encode using the minimization statement of `metasp`. For the preferred semantics we take the module π_{adm} and minimize the $\mathrm{out}/1$ predicate. This in turn gives us the subset-maximal admissible extensions, which captures the definition of preferred semantics. The encodings for the semi-stable and stage semantics are similar. Here we minimize the predicate $\mathrm{not\_in\_range}/1$ from the π_{range} module.

$$\begin{aligned}\pi_{prf\_metasp} &= \pi_{adm} \cup \{\#\mathit{minimize}[\mathrm{out}]\}.\\ \pi_{sem\_metasp} &= \pi_{adm} \cup \pi_{range} \cup \{\#\mathit{minimize}[\mathrm{not\_in\_range}]\}.\\ \pi_{stg\_metasp} &= \pi_{cf} \cup \pi_{range} \cup \{\#\mathit{minimize}[\mathrm{not\_in\_range}]\}.\end{aligned}$$

The following results follow now quite directly.

Proposition 3. *For any AF F, we have*

1. *$prf(F) \cong \mathcal{AS}(\pi_{prf\_metasp}(\hat{F}))$,*
2. *$sem(F) \cong \mathcal{AS}(\pi_{sem\_metasp}(\hat{F}))$, and*
3. *$stg(F) \cong \mathcal{AS}(\pi_{stg\_metasp}(\hat{F}))$.*

Next we give two different encodings for computing resolution-based grounded extensions. Both encodings use subset minimization for the resolution part, i.e. the resulting extension is subset minimal with respect to all possible resolutions. The first one computes the grounded extension for the guessed resolution explicitly (adapting the encoding from [11]; instead of the defeat predicate we use $\mathrm{defeat\_minus\_beta}$, since we need the grounded extensions of a restricted defeat relation). In fact, the π_{res} module which we give next guesses this restricted defeat relation $\{R \setminus \beta\}$ for a resolution β.

$$\begin{aligned}\pi_{res} = \{\ &\mathrm{defeat\_minus\_beta}(X,Y) \leftarrow \mathrm{defeat}(X,Y), \mathit{not}\ \mathrm{defeat\_minus\_beta}(Y,X),\\ &\qquad X \neq Y;\\ &\mathrm{defeat\_minus\_beta}(X,Y) \leftarrow \mathrm{defeat}(X,Y), \mathit{not}\ \mathrm{defeat}(Y,X);\\ &\mathrm{defeat\_minus\_beta}(X,X) \leftarrow \mathrm{defeat}(X,X)\}.\end{aligned}$$

The second encoding uses the `metasp` subset minimization additionally to get the grounded extension from the complete extensions of the current resolution (recall that the grounded extension is in fact the unique subset-minimal complete extension). We again use the restricted defeat relation.

$$\pi_{com} = \pi_{adm} \cup \{\ \mathrm{undefended}(X) \leftarrow \mathrm{defeat\_minus\_beta}(Y, X), \mathit{not}\, \mathrm{defeated}(Y);$$
$$\leftarrow \mathrm{out}(X), \mathit{not}\, \mathrm{undefended}(X)\ \}.$$

Now we can give the two encodings for resolution-based grounded semantics.

$$\pi_{grd^*\_metasp} = \pi_{grd} \cup \pi_{res} \cup \{\#minimize[\mathrm{in}]\}$$
$$\pi'_{grd^*\_metasp} = \pi_{com} \cup \pi_{res} \cup \{\#minimize[\mathrm{in}]\}.$$

Proposition 4. *For any AF F and $\pi \in \{\pi_{grd^*\_metasp}, \pi'_{grd^*\_metasp}\}$, $grd^*(F)$ corresponds to $\mathcal{AS}(\pi(\hat{F}))$ in the sense of Definition 5, but without property (iii).*

3.3 Alternative Encodings for Resolution-based Grounded Semantics

So far, we have shown two encodings for the resolution-based grounded semantics via optimization programs, i.e. we made use of the $\#minimize$ statement under the subset-inclusion semantics. From the complexity point of view this is not adequate, since we expressed a problem on the NP-layer (see Table 1) via an encoding which implicitly makes use of disjunction (see Table 2 for the actual complexity of optimization programs). Hence, we provide here an alternative encoding for the resolution-based grounded semantics based on the verification algorithm proposed by Baroni *et al.* in [1]. This encoding is just a normal program and thus located at the right level of complexity.

We need some further notation. For an AF $F = (A, R)$ and a set $S \subseteq A$ we define $F|_S = ((A \cap S), R \cap (S \times S))$ as the *sub-framework* of F wrt S; furthermore we also use $F - S$ as a shorthand for $F|_{A \setminus S}$. By $SCCs(F)$, we denote the set of strongly connected components of an AF $F = (A, R)$ which identify the vertices of a maximal strongly connected[4] subgraphs of F; $SCCs(F)$ is thus a partition of A. A partial order $\prec_F$ over $SCCs(F) = \{C_1, \ldots, C_n\}$, denoted as $(C_i \prec_F C_j)$ for $i \neq j$, is defined, if $\exists x \in C_i, y \in C_j$ such that there is a directed path from x to y in F.

Definition 6. *A $C \in SCCs(F)$ is* minimal relevant *(in an AF F) iff C is a minimal element of $\prec_F$ and $F|_C$ satisfies the following:*

(a) the attack relation $R(F|_C)$ of F is irreflexive, i.e. $(x, x) \notin R(F|_C)$ for all arguments x;
(b) $R(F|_C)$ is symmetric, i.e. $(x, y) \in R(F|_C) \Leftrightarrow (y, x) \in R(F|_C)$;
(c) the undirected graph *obtained by replacing each (directed) pair $\{(x, y), (y, x)\}$ in $F|_C$ with a single undirected edge $\{x, y\}$ is acyclic.*

The set of minimal relevant SCCs in F is denoted by $MR(F)$.

Proposition 5 ([1]). *Given an AF $F = (A, R)$ such that $(F - S_R^+) \neq (\emptyset, \emptyset)$ and $MR(F - S_R^+) \neq \emptyset$, where $S = grd(F)$, a set $U \subseteq A$ of arguments is* resolution-based grounded *in F, i.e. $U \in grd^*(F)$ iff the following conditions hold:*

(i) $U \cap S_R^+ = S$;
(ii) $(T \cap \Pi_F) \in stb(F|_{\Pi_F})$, where $T = U \setminus S_R^+$, and $\Pi_F = \bigcup_{V \in MR(F - S_R^+)} V$;
(iii) $(T \cap \Pi_F^C) \in grd^*(F|_{\Pi_F^C} - (S_R^+ \cup (T \cap \Pi_F)_R^{\oplus}))$, where T and Π_F are as in (ii) and $\Pi_F^C = A \setminus \Pi_F$.

To illustrate the conditions of Proposition 5, let us have a look at our example.

[4] A directed graph is called *strongly connected* if there is a directed path from each vertex in the graph to every other vertex of the graph.

Example 4. Consider the AF F of Example 1. Let us check whether $U = \{a, d, f\}$ is resolution-based grounded in F, i.e. whether $U \in grd^*(F)$. $S = \{a\}$ is the grounded extension of F and $S_R^+ = \{a, b\}$, hence the first Condition (i) is satisfied. We obtain $T = \{d, f\}$ and $\Pi_F = \{c, d\}$. We observe that $T \cap \Pi_F = \{d\}$ is a stable extension of the AF $F|_{\Pi_F}$; that satisfies Condition (ii). Now we need to check Condition (iii); we first identify the necessary sets: $\Pi_F^C = \{a, b, e, f\}$, $T \cap \Pi_F^C = \{f\}$ and $(T \cap \Pi_F)_R^{\oplus} = \{c, e\}$. It remains to check $\{f\} \in grd^*(\{f\}, \emptyset)$ which is easy to see. Hence, $U \in grd^*(F)$.

The following encoding is based on the Guess&Check procedure which was also used for the encodings in [11]. After guessing all conflict-free sets with the program π_{cf}, we check whether the conditions of Definition 6 and Proposition 5 hold. Therefore the program π_{arg_set} makes a copy of the actual arguments, defeats and the guessed set to the predicates $\mathrm{arg\_set}/2, \mathrm{defeatN}/3$ and $\mathrm{inU}/2$. The first variable in these three predicates serves as an identifier for the iteration of the algorithm (this is necessary to handle the recursive nature of Proposition 5). In all following predicates we will use the first variable of each predicate like this. As in some previous encodings in this paper, we use the program $\pi_<$ to obtain an order over the arguments, and we start our computation with the infimum represented by the predicate $\mathrm{inf}/1$.

$$\begin{aligned}\pi_{arg\_set} = \{\ & \mathrm{arg\_set}(N, X) \leftarrow \mathrm{arg}(X), \mathrm{inf}(N);\\ & \mathrm{inU}(N, X) \leftarrow \mathrm{in}(X), \mathrm{inf}(N);\\ & \mathrm{defeatN}(N, Y, X) \leftarrow \mathrm{arg\_set}(N, X), \mathrm{arg\_set}(N, Y), \mathrm{defeat}(Y, X)\ \}.\end{aligned}$$

We use here the program $\pi_{defendedN}$ (which is a slight variant of the program $\pi_{defended}$) together with the program $\pi_{groundN}$ where we perform a fixed-point computation of the predicate $\mathrm{defendedN}/2$, but now we use an additional argument N for the iteration step where predicates $\mathrm{arg\_set}/2$, $\mathrm{defeatN}/3$ and $\mathrm{inS}/2$ replace $\mathrm{arg}/1$, $\mathrm{defeat}/2$ and $\mathrm{in}/1$. In $\pi_{groundN}$ we then obtain the predicate $\mathrm{inS}(N, X)$ which identifies argument X to be in the grounded extension of the iteration N.

$$\pi_{groundN} = \pi_{cf} \cup \pi_< \cup \pi_{arg\_set} \cup \pi_{defendedN} \cup \{\ \mathrm{inS}(N, X) \leftarrow \mathrm{defendedN}(N, X)\ \}.$$

The next module $\pi_{F\_minus\_range}$ computes the arguments in $(F - S_R^+)$, represented by the predicate $\mathrm{notInSplusN}/2$, via predicates $\mathrm{in\_SplusN}/2$ and $\mathrm{u\_cap\_Splus}/2$ (for S_R^+ and $U \cap S_R^+$). The two constraints check condition (i) of Proposition 5.

$$\begin{aligned}\pi_{F\_minus\_range} = \{\ & \mathrm{in\_SplusN}(N, X) \leftarrow \mathrm{inS}(N, X);\\ & \mathrm{in\_SplusN}(N, X) \leftarrow \mathrm{inS}(N, Y), \mathrm{defeatN}(N, Y, X);\\ & \mathrm{u\_cap\_Splus}(N, X) \leftarrow \mathrm{inU}(N, X), \mathrm{in\_SplusN}(N, X);\\ & \leftarrow \mathrm{u\_cap\_Splus}(N, X), \mathit{not}\ \mathrm{inS}(N, X);\\ & \leftarrow \mathit{not}\ \mathrm{u\_cap\_Splus}(N, X), \mathrm{inS}(N, X);\\ & \mathrm{notInSplusN}(N, X) \leftarrow \mathrm{arg\_set}(N, X), \mathit{not}\ \mathrm{in\_SplusN}(N, X)\ \}.\end{aligned}$$

The module π_{MR} computes $\Pi_F = \bigcup_{V \in MR(F - S_R^+)} V$, where $\mathrm{mr}(N, X)$ denotes that an argument is contained in a set $V \in MR$. Therefore we need to check all three conditions of Definition 6. The first two rules compute the predicate $\mathrm{reach}(N, X, Y)$ if there is a path between the arguments $X, Y \in (F - S_R^+)$. With this predicate we will identify the SCCs. The third rule computes $\mathrm{self\_defeat}/2$ for all arguments violating Condition (a). Next we need to check Condition (b). With $\mathrm{nsym}/2$ we obtain those arguments which do not have a symmetric attack to any other argument from the same component. Condition (c) is a bit more tricky. With predicate $\mathrm{reachnotvia}/4$ we say that there is a path from X to Y not going over argument V in the framework $(F - S_R^+)$. With this predicate at hand we can check for cycles with $\mathrm{cyc}/4$. Then, to complete Condition (c) we derive $\mathrm{bad}/2$ for all arguments which are connected to a cycle (or a self-defeating argument). In the predicate $\mathrm{pos\_mr}/2$, we put all the three conditions together and say that an argument x is possibly in a set $V \in MR$ if (i) $x \in (F - S_R^+)$, (ii) x is neither connected to a cycle nor self-defeating, and (iii) for all y it holds that $(x, y) \in (F - S_R^+) \Leftrightarrow (y, x) \in (F - S_R^+)$. Finally we

only need to check if the SCC obtained with $\mathrm{pos\_mr}/2$ is a minimal element of $\prec_F$. Hence we get with $\mathrm{notminimal}/2$ all arguments not fulfilling this, and in the last rule we obtain with $\mathrm{mr}/2$ the arguments contained in a minimal relevant SCC.

$$
\begin{aligned}
\pi_{MR} = \{\ & \mathrm{reach}(N,X,Y) \leftarrow \mathrm{notInSplusN}(N,X), \mathrm{notInSplusN}(N,Y), \mathrm{defeatN}(N,X,Y);\\
& \mathrm{reach}(N,X,Y) \leftarrow \mathrm{notInSplusN}(N,X), \mathrm{defeatN}(N,X,Z), \mathrm{reach}(N,Z,Y),\\
& \qquad X! = Y;\\
& \mathrm{self\_defeat}(N,X) \leftarrow \mathrm{notInSplusN}(N,X), \mathrm{defeatN}(N,X,X);\\
& \mathrm{nsym}(N,X) \leftarrow \mathrm{notInSplusN}(N,X), \mathrm{notInSplusN}(N,Y), \mathrm{defeatN}(N,X,Y),\\
& \qquad \mathit{not}\ \mathrm{defeatN}(N,Y,X), \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X), X! = Y;\\
& \mathrm{nsym}(N,Y) \leftarrow \mathrm{notInSplusN}(N,X), \mathrm{notInSplusN}(N,Y), \mathrm{defeatN}(N,X,Y),\\
& \qquad \mathit{not}\ \mathrm{defeatN}(N,Y,X), \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X), X! = Y;\\
& \mathrm{reachnotvia}(N,X,V,Y) \leftarrow \mathrm{defeatN}(N,X,Y), \mathrm{notInSplusN}(N,V),\\
& \qquad \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X), X! = V, Y! = V;\\
& \mathrm{reachnotvia}(N,X,V,Y) \leftarrow \mathrm{reachnotvia}(N,X,V,Z), \mathrm{reach}(N,X,Y),\\
& \qquad \mathrm{reachnotvia}(N,Z,V,Y), \mathrm{reach}(N,Y,X),\\
& \qquad Z! = V, X! = V, Y! = V;\\
& \mathrm{cyc}(N,X,Y,Z) \leftarrow \mathrm{defeatN}(N,X,Y), \mathrm{defeatN}(N,Y,X),\\
& \qquad \mathrm{defeatN}(N,Y,Z), \mathrm{defeatN}(N,Z,Y),\\
& \qquad \mathrm{reachnotvia}(N,X,Y,Z), X! = Y, Y! = Z, X! = Z;\\
& \mathrm{bad}(N,Y) \leftarrow \mathrm{cyc}(N,X,U,V), \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X);\\
& \mathrm{bad}(N,Y) \leftarrow \mathrm{self\_defeat}(N,X), \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X);\\
& \mathrm{pos\_mr}(N,X) \leftarrow \mathrm{notInSplusN}(N,X), \mathit{not}\ \mathrm{bad}(N,X), \mathit{not}\ \mathrm{self\_defeat}(N,X),\\
& \qquad \mathit{not}\ \mathrm{nsym}(N,X);\\
& \mathrm{notminimal}(N,Z) \leftarrow \mathrm{reach}(N,X,Y), \mathrm{reach}(N,Y,X),\\
& \qquad \mathrm{reach}(N,X,Z), \mathit{not}\ \mathrm{reach}(N,Z,X);\\
& \mathrm{mr}(N,X) \leftarrow \mathrm{pos\_mr}(N,X), \mathit{not}\ \mathrm{notminimal}(N,X)\ \}.
\end{aligned}
$$

We now turn to Condition (ii) of Proposition 5, where the first rule in $\pi_{stableN}$ computes the set $T = U \setminus S_R^+$. Then we check whether $T = \emptyset$ and $MR(F - S_R^+) = \emptyset$ via predicates $\mathrm{emptyT}/1$ and $\mathrm{not\_exists\_mr}/1$. If this is so, we terminate the iteration in the last module $\pi_{iterate}$. The first constraint eliminates those guesses where $MR(F - S_R^+) = \emptyset$ but $T \neq \emptyset$, because the algorithm is only defined for AFs fulfilling this. Finally we derive the arguments which are defeated by the set T in the MR denoted by $\mathrm{defeated}/2$, and with the last constraint we eliminate those guesses where there is an argument not contained in T and not defeated by T in MR and hence $(T \cap \mathit{\Pi}_F) \notin \mathit{stb}(F|_{\mathit{\Pi}_F})$.

$$
\begin{aligned}
\pi_{stableN} = \{\ & \mathrm{t}(N,X) \leftarrow \mathrm{inU}(N,X), \mathit{not}\ \mathrm{inS}(N,X);\\
& \mathrm{nemptyT}(N) \leftarrow \mathrm{t}(N,X);\\
& \mathrm{emptyT}(N) \leftarrow \mathit{not}\ \mathrm{nemptyT}(N), \mathrm{arg\_set}(N,X);\\
& \mathrm{existsMR}(N) \leftarrow \mathrm{mr}(N,X), \mathrm{notInSplusN}(N,X);\\
& \mathrm{not\_exists\_mr}(N) \leftarrow \mathit{not}\ \mathrm{existsMR}(N), \mathrm{notInSplusN}(N,X);\\
& \mathrm{true}(N) \leftarrow \mathrm{emptyT}(N), \mathit{not}\ \mathrm{existsMR}(N);\\
& \leftarrow \mathrm{not\_exists\_mr}(N), \mathrm{nemptyT}(N);\\
& \mathrm{defeated}(N,X) \leftarrow \mathrm{mr}(N,X), \mathrm{mr}(N,Y), \mathrm{t}(N,Y), \mathrm{defeatN}(N,Y,X);\\
& \leftarrow \mathit{not}\ \mathrm{t}(N,X), \mathit{not}\ \mathrm{defeated}(N,X), \mathrm{mr}(N,X)\ \}.
\end{aligned}
$$

With the last module $\pi_{iterate}$ we perform Step (iii) of Proposition 5. The predicate $\mathrm{t\_mrOplus}/2$ computes the set $(T \cap \Pi_F)^{\oplus}_R$ and with the second rule we start the next iteration for the framework $(F|_{\Pi_F^C} - (S_R^+ \cup (T \cap \Pi_F)^{\oplus}_R))$ and the set $(T \cap \Pi_F^C)$.

$$\begin{aligned}\pi_{iterate} = \{\ & \mathrm{t\_mrOplus}(N,Y) \leftarrow \mathrm{t}(N,X), \mathrm{mr}(N,X), \mathrm{defeatN}(N,X,Y);\\ & \mathrm{arg\_set}(M,X) \leftarrow \mathrm{notInSplusN}(N,X), \mathit{not}\ \mathrm{mr}(N,X),\\ & \qquad\qquad \mathit{not}\ \mathrm{t\_mrOplus}(N,X), \mathrm{succ}(N,M), \mathit{not}\ \mathrm{true}(N);\\ & \mathrm{inU}(M,X) \leftarrow \mathrm{t}(N,X), \mathit{not}\ \mathrm{mr}(N,X), \mathrm{succ}(N,M), \mathit{not}\ \mathrm{true}(N)\ \}.\end{aligned}$$

Finally we put everything together and obtain the program π_{grd^*}.

$$\pi_{grd^*} = \pi_{groundN} \cup \pi_{F\_minus\_range} \cup \pi_{MR} \cup \pi_{stableN} \cup \pi_{iterate}.$$

Proposition 6. *For any AF F, $grd^*(F) \cong \mathcal{AS}(\pi_{grd^*}(\hat{F}))$.*

4 Experimental Evaluation

In this section we present our results of the performance evaluation. We compared the time needed for computing all extensions for the semantics described earlier using both the handcraft saturation-based and the alternative `metasp` encodings.

The tests were executed on an openSUSE based machine with eight Intel Xeon processors (2.33 GHz) and 49 GB memory. For computing the answer sets, we used `gringo` (version 3.0.3) for grounding and the solver `claspD` (version 1.1.1). The latter being the variant for disjunctive answer-set programs.

We randomly generated AFs (i.e. graphs) ranging from 20 to 110 arguments. We used two parametrized methods for generating the attack relation.The first generates arbitrary AFs and inserts for any pair (a, b) the attack from a to b with a given probability p. The other method generates AFs with a $n \times m$ grid-structure. We consider two different neighborhoods, one connecting arguments vertically and horizontally and one that additionally connects the arguments diagonally. Such a connection is a mutual attack with a given probability p and in only one direction otherwise. The probability p was chosen between 0.1 and 0.4.

Overall 14388 tests were executed, with a timeout of five minutes for each execution. Timed out instances are considered as solved in 300 seconds. The time consumption was measured using the Linux `time` command. For all the tests we let the solver generate all answer sets, but only outputting the number of models. To minimize external influences on the test runs, we alternated the different encodings during the tests.

Figures 1 - 3 depict the results for the preferred, semi-stable and stage semantics respectively. The figures show the average computation time for both the handcraft and the `metasp` encoding for a certain number of arguments. We distinguish here between arbitrary, i.e. completely random AFs and grid structured ones. One can see that the `metasp` encodings have a better performance, compared to the handcraft encodings. In particular, for the stage semantics the performance difference between the handcraft and the `metasp` variant is noticeable. Recall that the average computation time includes the timeouts, which strongly influence the diagrams.

For the resolution-based grounded semantics Figure 4 shows again the average computation time needed for a certain number of arguments. Let us first consider the case of arbitrary AFs. The handcraft encoding struggled with AFs of size 40 or larger. Many of those instances could not be solved due to memory faults. This is indicated by the missing data points. Both `metasp` encodings performed better overall, but still many timeouts were encountered. If we look more closely at the structured AFs then we see that $\pi'_{grd^*\_metasp}$ performs better overall than the other `metasp` variant. Interestingly, computing the grounded part with a handcraft encoding without a Guess&Check part did not result in a lower computation time on average. The handcraft encoding performed better than $\pi_{grd^*\_metasp}$ on grids.

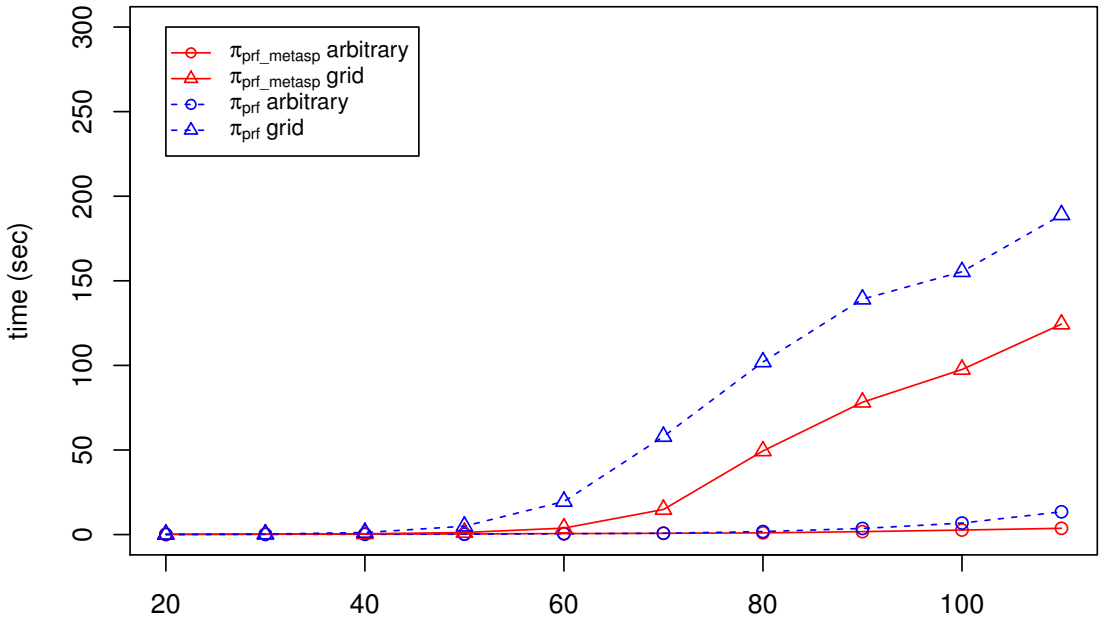


Fig. 1. Average computation time for preferred semantics.

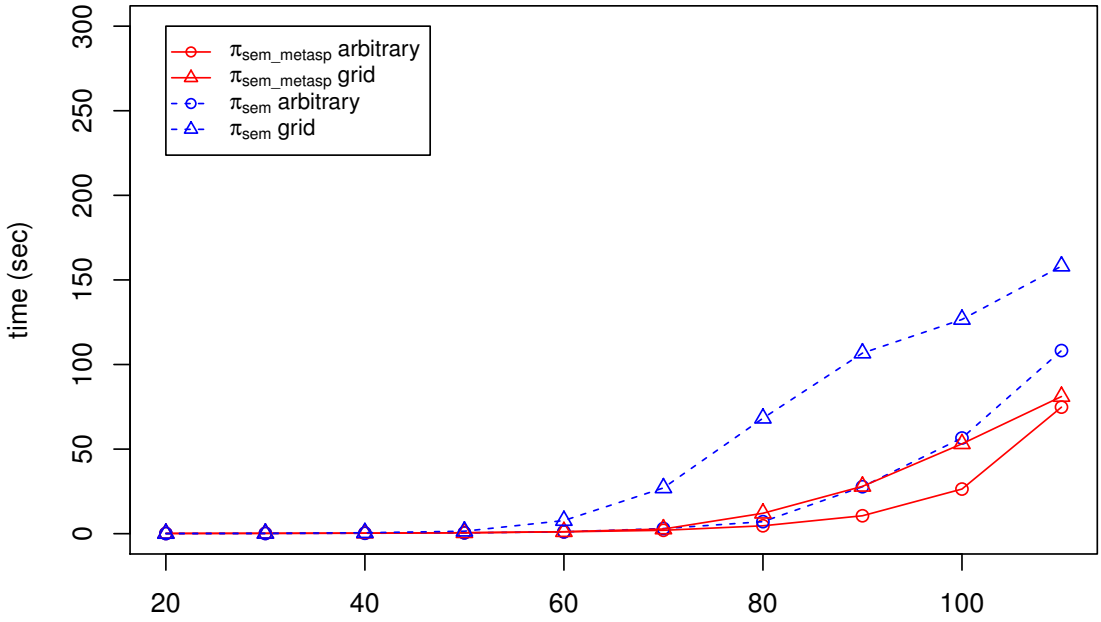


Fig. 2. Average computation time for semi-stable semantics.

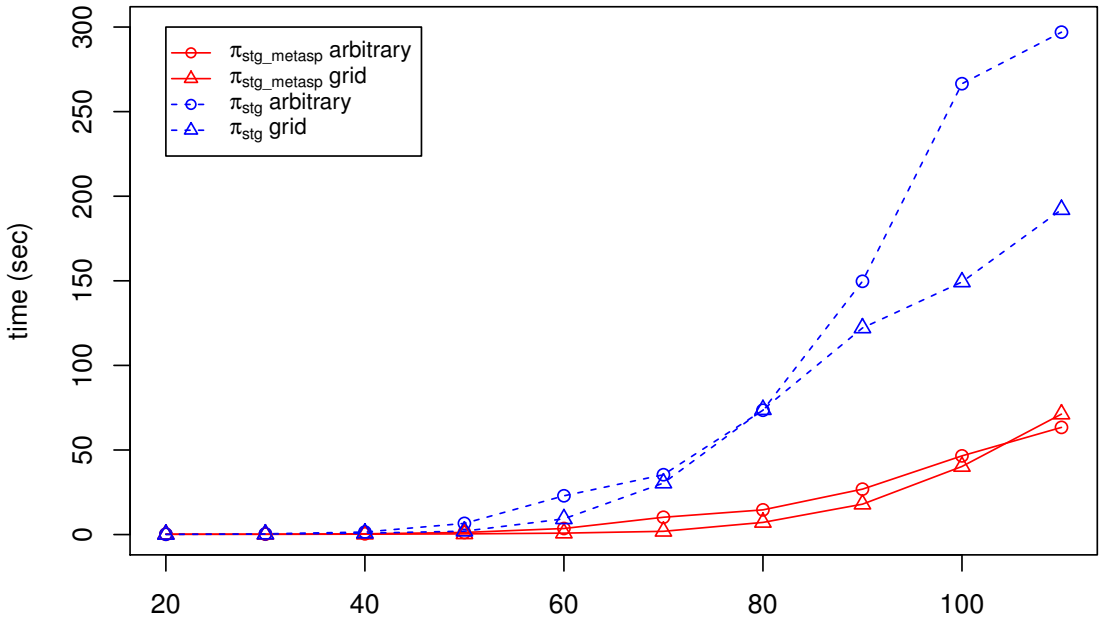


Fig. 3. Average computation time for stage semantics.

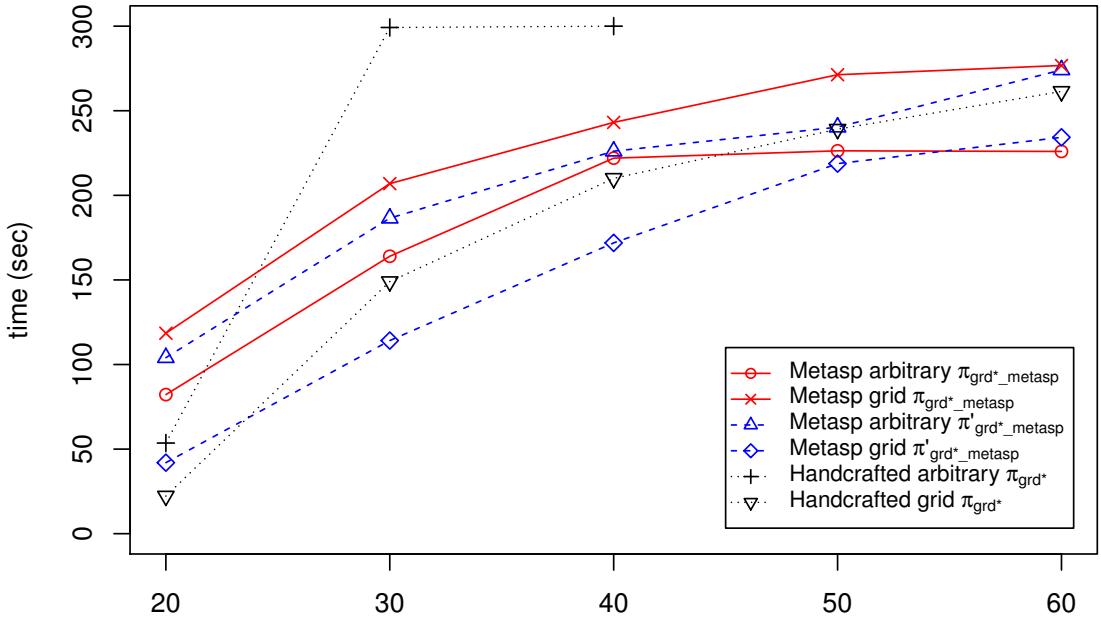


Fig. 4. Average computation time for resolution-based grounded semantics

5 Conclusion

In this paper, we inspected various ASP encodings for four prominent semantics in the area of abstract argumentation. (1) For the preferred and the semi-stable semantics, we compared existing saturation-based encodings [11] (here we called them handcrafted encodings) with novel alternative encodings which are based on the recently developed `metasp` approach [13], where subset minimization can be directly specified (and a front-end, i.e. a meta-interpreter) compiles such statements back into the core ASP language. (2) For the stage semantics, we presented here both a handcrafted and a `metasp` encoding. Finally, (3) for the resolution-based grounded semantics we provided three encodings, two of them using the `metasp` techniques.

Although the `metasp` encodings are much simpler to design (since saturation techniques are delegated to the meta-interpreter), they perform surprisingly well when compared with the handcraft encodings which are directly given to the ASP solver. This shows the practical relevance of the `metasp` technique also in the area of abstract argumentation. Future work has to focus on further experiments which hopefully will strengthen our observations.

References

1. P. Baroni, P. E. Dunne, and M. Giacomin. On the resolution-based family of abstract argumentation semantics and its grounded instance. *Artif. Intell.*, 175(3-4):791–813, 2011.
2. P. Baroni and M. Giacomin. Semantics of abstract argument systems. In I. Rahwan and G. Simari, editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer, 2009.
3. T. J. M. Bench-Capon and P. E. Dunne. Argumentation in artificial intelligence. *Artif. Intell.*, 171(10-15):619–641, 2007.
4. M. Caminada. Semi-stable semantics. In Proc. *COMMA 2006*, pages 121–130, 2006.
5. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
6. Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, 170(1-2):209–244, 1996.
7. P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
8. P. E. Dunne and T. J. M. Bench-Capon. Coherence in finite argument systems. *Artif. Intell.*, 141(1/2):187–203, 2002.
9. P. E. Dunne and M. Caminada. Computational complexity of semi-stable semantics in abstract argumentation frameworks. In Proc. *JELIA 2008*, pages 153–165, 2008.
10. W. Dvořák and S. Woltran. Complexity of semi-stable and stage semantics in argumentation frameworks. *Inf. Process. Lett.*, 110(11):425–430, 2010.
11. U. Egly, S. A. Gaggl, and S. Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.
12. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.
13. M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11(4-5): 821–839 (2011).
14. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.
15. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
16. F. Toni and M. Sergot. Argumentation and answer set programming. In M. Balduccini and T.C. Son, editors, *Gelfond Festschrift*, volume 6565 of *LNAI*, pages 164–180. Springer, 2011.
17. B. Verheij. Two approaches to dialectical argumentation: admissible sets and argumentation stages. In Proc. *NAIC'96*, pages 357–368, 1996.

Confidentiality-Preserving Data Publishing for Credulous Users by Extended Abduction

Katsumi Inoue[1], Chiaki Sakama[2] and Lena Wiese[1⋆]

[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
{ki|wiese}@nii.ac.jp
[2] Department of Computer and Communication Sciences Wakayama University
930 Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. Publishing private data on external servers incurs the problem of how to avoid unwanted disclosure of confidential data. We study a problem of confidentiality in extended disjunctive logic programs and show how it can be solved by extended abduction. In particular, we analyze how credulous non-monotonic reasoning affects confidentiality.

Keywords: Data publishing, confidentiality, privacy, extended abduction, answer set programming, negation as failure, non-monotonic reasoning

1 Introduction

Confidentiality of data (also called privacy or secrecy in some contexts) is a major security goal. Releasing data to a querying user without disclosing confidential information has long been investigated in areas like access control, k-anonymity, inference control, and data fragmentation. Such approaches prevent disclosure according to some security policy by restricting data access (denial, refusal), by modifying some data (perturbation, noise addition, cover stories, lying, weakening), or by breaking sensitive associations (fragmentation). Several approaches (like [3, 8, 13, 14, 2, 15]) employ logic-based mechanisms to ensure data confidentiality. In particular, [5] use brave reasoning in default logic theories to solve a privacy problem in a classical database (a set of ground facts). For a non-classical knowledge base (where negation as failure not is allowed) [16] study correctness of access rights. Confidentiality of predicates in collaborative multi-agent abduction is a topic in [10].

In this article we analyze **confidentiality-preserving data publishing** in a knowledge base setting: data as well as integrity constraints or deduction rules are represented as logical formulas. If such a knowledge base is released to the public for general querying (e.g., microcensus data) or outsourced to a storage provider (e.g., database-as-a-service in cloud computing), confidential data could be disclosed. We assume that users accessing the published knowledge base use a form of credulous (also called brave) reasoning to retrieve data from it; users also possess some invariant "a priori knowledge" that can be applied to these data to deduce further information. On the knowledge base side, a confidentiality policy specifies which is the confidential information that must never be disclosed. This paper is one of only few papers (see [11, 16, 10]) covering confidentiality for logic programs. This formalism however has relevance in multi-agent communications where agent knowledge is modeled by logic programs. With **extended abduction** ([12]) we obtain a "secure version" of the knowledge base that can safely be published even when a priori knowledge is applied. We show that computing the secure version for a credulous user corresponds to finding a skeptical anti-explanation for all the elements of the confidentiality policy. Extended abduction has been used in different applications like for example providing a logical framework for dishonest reasoning [11]. It can be solved by computing the answer sets of an update program (see [12]); thus an implementation of extended abduction can profit from current answer set programming (ASP) solvers [4]. To retrieve the

⋆ Lena Wiese gratefully acknowledges a postdoctoral research grant of the German Academic Exchange Service (DAAD).

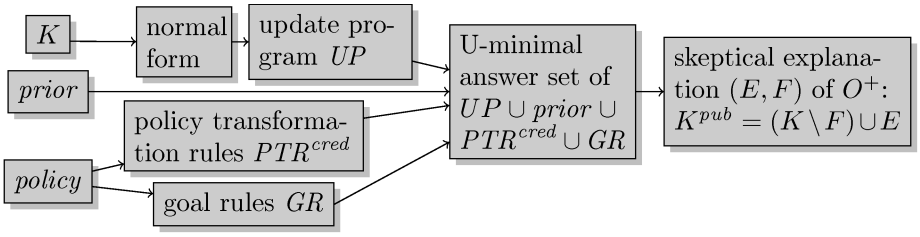


Fig. 1. Finding a confidentiality-preserving K^{pub} for a credulous user

confidentiality-preserving knowledge base K^{pub} from the input knowledge base K, the a priori knowledge $prior$ and the confidentiality policy $policy$, a row of transformations are applied; the overall approach is depicted in Figure 1.

In sum, this paper makes the following contributions:

- it formalizes confidentiality-preserving data publishing for a user who retrieves data under a credulous query response semantics.
- it devises a procedure to securely publish a logic program (with an expressiveness up to extended disjunctive logic programs) respecting a subset-minimal change semantics.
- it shows that confidentiality-preservation for credulous users corresponds to finding a skeptical anti-explanation and can be solved by extended abduction.

In the remainder of this article, Section 2 provides background on extended disjunctive logic programs and answer set semantics; Section 3 defines the problem of confidentiality in data publishing; Section 4 recalls extended abduction and update programs; Section 5 shows how answer sets of update programs correspond to confidentiality-preserving knowledge bases; and Section 6 gives some discussion and concluding remarks.

2 EDPs and answer set semantics

In this article, a knowledge base K is represented by an *extended disjunctive logic program* (EDP) – a set of formulas called *rules* of the form:

$$L_1; \ldots; L_l \leftarrow L_{l+1}, \ldots, L_m, notL_{m+1}, \ldots, notL_n \quad (n \geq m \geq l \geq 0)$$

A rule contains literals L_i, disjunction ";", conjunction ",", negation as failure "not", and material implication "$\leftarrow$". A literal is a first-order atom or an atom preceded by classical negation "$\neg$". $notL$ is called a *NAF-literal*. The disjunction left of the implication $\leftarrow$ is called the *head*, while the conjunction right of $\leftarrow$ is called the *body* of the rule. For a rule R, we write $head(R)$ to denote the set of literals $\{L_1, \ldots, L_l\}$ and $body(R)$ to denote the set of (NAF-)literals $\{L_{l+1}, \ldots, L_m, notL_{m+1}, \ldots, notL_n\}$. Rules consisting only of a singleton head $L \leftarrow$ are identified with the literal L and used interchangeably. An EDP is ground if it contains no variables. If an EDP contains variables, it is identified with the set of its ground instantiations: the elements of its Herbrand universe are substituted in for the variables in all possible ways. We assume that the language contains no function symbol, so that each rule with variables represents a finite set of ground rules. For a program K, we denote $\mathscr{L}_K$ the set of ground literals in the language of K. Note that EDPs offer a high expressiveness including disjunctive and non-monotonic reasoning.

Example 1. In a medical knowledge base *Ill*(x, y) states that a patient x is ill with disease y; *Treat*(x, y) states that x is treated with medicine y. Assume that if you read the record and find that one treatment (Medi1) is recorded and another one (Medi2) is not recorded, then you know that the patient is at least ill with Aids or Flu (and possibly has other illnesses).
$K = \{$*Ill*$(x,$ Aids$)$; *Ill*$(x,$ Flu$) \leftarrow$ *Treat*$(x,$ Medi1$)$, not*Treat*$(x,$ Medi2$)$,
Ill$($Mary, Aids$)$, *Treat*$($Pete, Medi1$)\}$ serves as a running example.

The semantics of K can be given by the answer set semantics [7]: A set $S \subseteq \mathscr{L}_K$ of ground literals *satisfies* a ground literal L if $L \in S$; S satisfies a conjunction if it satisfies every conjunct; S satisfies a disjunction if it satisfies at least one disjunct; S satisfies a ground rule if whenever the body literals are contained in S ($\{L_{l+1}, \ldots, L_m\} \subseteq S$) and all NAF-literals are not contained in S ($\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$), then at least one head literal is contained in S ($L_i \in S$ for an i such that $1 \leq i \leq l$). If an EDP K contains no NAF-literals ($m = n$), then such a set S is an *answer set* of K if S is a subset-minimal set such that

1. S satisfies every rule from the ground instantiation of K,
2. If S contains a pair of complementary literals L and $\neg L$, then $S = \mathscr{L}_K$.

This definition of an answer set can be extended to full EDPs (containing NAF-literals) as in [12]: For an EDP K and a set of ground literals $S \subseteq \mathscr{L}_K$, K can be transformed into a NAF-free program K^S as follows. For every ground rule from the ground instantiation of K (with respect to its Herbrand universe), the rule $L_1; \ldots; L_l \leftarrow L_{l+1}, \ldots, L_m$ is in K^S if $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$. Then, S is an answer set of K if S is an answer set of K^S. An answer set is *consistent* if it is not $\mathscr{L}_K$. A program K is *consistent* if it has a consistent answer set; otherwise K is *inconsistent*.

Example 2. The example K has the following two consistent answer sets

$$S_1 = \{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})\}$$
$$S_2 = \{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})\}$$

When adding the negative fact $\neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})$ to K, then there is just one consistent answer set left: for $K' := K \cup \{\neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})\}$ the unique answer set is

$$S' = \{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu}), \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})\}.$$

If a rule R is satisfied in *every* answer set of K, we write $K \models R$. In particular, $K \models L$ if a literal L is included in every answer set of K.

3 Confidentiality-Preserving Knowledge Bases

When publishing a knowledge base K while preserving confidentiality of some data in K we do this according to

- the query response semantics that a user querying the published knowledge base applies; we focus on credulous query response semantics
- a confidentiality policy (denoted $policy$) describing confidential information that should not be released to the public
- background (a priori) knowledge (denoted $prior$) that a user can combine with query responses from the published knowledge base

First we define the credulous query response semantics: a ground formula Q is $true$ in K, if Q is satisfied in some answer set of K – that is, there might be answer sets that do not satisfy Q. If a rule Q is non-ground and contains some free variables, the credulous response of K is the set of ground instantiations of Q that are $true$ in K.

Definition 1 (Credulous query response semantics). *Let U be the Herbrand universe of a consistent knowledge base K. The* credulous query responses *of formula $Q(X)$ (with a vector X of free variables) in K are*

$$\mathit{cred}(K, Q(X)) = \{Q(A) \mid A \text{ is a vector of elements } a \in U \text{ and there is an answer set of } K \text{ that satisfies } Q(A)\}$$

In particular, for a ground formula Q,

$$\mathit{cred}(K, Q) = \begin{cases} Q & \text{if } K \text{ has an answer set that satisfies } Q \\ \emptyset & \text{otherwise} \end{cases}$$

It is usually assumed that in addition to the query responses a user has some additional knowledge that he can apply to the query responses. Hence, we additionally assume given a set of rules as some *invariant* **a priori knowledge** $prior$. Without loss of generality we assume that $prior$ is an EDP. Thus, the priori knowledge may consist of additional facts that the user assumes to hold in K, or some rules that the user can apply to data in K to deduce new information.

A **confidentiality policy** $policy$ specifies confidential information. We assume that $policy$ contains only conjunctions of (NAF-)literals. However, see Section 5.1 for a brief discussion on how to use more expressive policy formulas. We do not only have to avoid that the published knowledge base contains confidential information but also prevent the user from deducing confidential information with the help of his a priori knowledge; this is known as the inference problem [6, 2].

Example 3. If we wish to declare the disease aids as confidential for any patient x we can do this with $policy = \{\mathit{Ill}(x, \mathsf{Aids})\}$. A user querying K^{pub} might know that a person suffering from flu is not able to work. Hence $prior = \{\neg \mathit{AbleToWork}(x) \leftarrow \mathit{Ill}(x, \mathsf{Flu})\}$. If we wish to also declare a lack of work ability as confidential, we can add this to the confidentiality policy: $policy' = \{\mathit{Ill}(x, \mathsf{Aids}) \ , \ \neg \mathit{AbleToWork}(x)\}$.

Next, we establish a definition of confidentiality-preservation that allows for the answer set semantics as an inference mechanism and respects the credulous query response semantics: when treating elements of the confidentiality policy as queries, the credulous responses must be empty.

Definition 2 (Confidentiality-preservation for credulous user). *A knowledge base K^{pub}* preserves confidentiality *of a given confidentiality policy under the credulous query response semantics and with respect to a given a priori knowledge $prior$, if for every conjunction $C(X)$ in the policy, the credulous query responses of $C(X)$ in $K^{pub} \cup prior$ are empty: $cred(K^{pub} \cup prior, C(X)) = \emptyset$.*

Note that in this definition the Herbrand universe of $K^{pub} \cup prior$ is applied in the query response semantics; hence, free variables in policy elements $C(X)$ are instantiated according to this universe. Note also that $K^{pub} \cup prior$ must be consistent. Confidentiality-preservation for *skeptical* query response semantics is topic of future work.

A goal secondary to confidentiality-preservation is minimal change: We want to publish as many data as possible and want to modify these data as little as possible. Different notions of minimal change are used in the literature (see for example [1] for a collection of minimal change semantics in a data integration setting). We apply a subset-minimal change semantics: we choose a K^{pub} that differs from K only subset-minimally. In other words, there is not other confidentiality-preserving knowledge base $K^{pub\prime}$ which inserts (or deletes) less rules to (from) K than K^{pub}.

Definition 3 (Subset-minimal change). *A confidentiality-preserving knowledge base K^{pub}* subset-minimally changes *K (or is* minimal*, for short) if there is no confidentiality-preserving knowledge base $K^{pub\prime}$ such that $((K \setminus K^{pub\prime}) \cup (K^{pub\prime} \setminus K)) \subset ((K \setminus K^{pub}) \cup (K^{pub} \setminus K))$.*

Example 4. For the example K and $policy$ and no a priori knowledge, the fact $\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})$ has to be deleted. But also $\mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})$ can be deduced credulously, because it is satisfied by answer set S_1. In order to avoid this, we have three options: delete $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})$, delete the non-literal rule in K or insert $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2})$. The same solutions are found for K, $policy'$ and $prior$: they block the credulous deduction of $\neg \mathit{AbleToWork}(\mathsf{Pete})$. The same applies to K' and $policy$.

In the following sections we obtain a minimal solution K^{pub} for a given input K, $prior$ and $policy$ by transforming the input into a problem of *extended abduction* and solving it with an appropriate update program.

4 Extended Abduction

Traditionally, given a knowledge base K and an observation formula O, *abduction* finds a "(positive) explanation" E – a set of hypothesis formulas – such that every answer set of the knowledge base and the explanation together satisfy the observation; that is, $K \cup E \models O$. Going beyond that [9, 12] use *extended*

abduction with the notions of "negative observations", "negative explanations" F and "anti-explanations". An abduction problem in general can be restricted by specifying a designated set $\mathcal{A}$ of *abducibles*. This set poses syntactical restrictions on the explanation sets E and F. In particular, positive explanations are characterized by $E \subseteq \mathcal{A} \setminus K$ and negative explanations by $F \subseteq K \cap \mathcal{A}$. If $\mathcal{A}$ contains a formula with variables, it is meant as a shorthand for all ground instantiations of the formula. In this sense, an EDP K accompanied by an EDP $\mathcal{A}$ is called an *abductive program* written as $\langle K, \mathcal{A}\rangle$. The aim of extended abduction is then to find (anti-)explanations as follows (where in this article only *skeptical* (anti-)explanations are needed):

- given a *positive* observation O, find a pair (E, F) where E is a positive explanation and F is a negative explanation such that
  1. **[skeptical explanation]** O is satisfied in *every* answer set of $(K \setminus F) \cup E$; that is, $(K \setminus F) \cup E \models O$
  2. **[consistency]** $(K \setminus F) \cup E$ is consistent
  3. **[abducibility]** $E \subseteq \mathcal{A} \setminus K$ and $F \subseteq \mathcal{A} \cap K$
- given a *negative* observation O, find a pair (E, F) where E is a positive anti-explanation and F is a negative anti-explanation such that
  1. **[skeptical anti-explanation]** there is *no* answer set of $(K \setminus F) \cup E$ in which O is satisfied
  2. **[consistency]** $(K \setminus F) \cup E$ is consistent
  3. **[abducibility]** $E \subseteq \mathcal{A} \setminus K$ and $F \subseteq \mathcal{A} \cap K$

Among (anti-)explanations, **minimal** (anti-)explanations characterize a subset-minimal alteration of the program K: an (anti-)explanation (E, F) of an observation O is called minimal if for any (anti-)explanation (E', F') of O, $E' \subseteq E$ and $F' \subseteq F$ imply $E' = E$ and $F' = F$.

For an abductive program $\langle K, \mathcal{A}\rangle$ both K and $\mathcal{A}$ are semantically identified with their ground instantiations with respect to the Herbrand universe, so that set operations over them are defined on the ground instances. Thus, when (E, F) contain formulas with variables, $(K \setminus F) \cup E$ means deleting every instance of formulas in F, and inserting any instance of formulas in E from/into K. When E contains formulas with variables, the set inclusion $E' \subseteq E$ is defined for any set E' of instances of formulas in E. Generally, given sets S and T of literals/rules containing variables, any set operation $\circ$ is defined as $S \circ T = inst(S) \circ inst(T)$ where $inst(S)$ is the ground instantiation of S. For example, when $p(x) \in T$, for any constant a occurring in T, it holds that $\{p(a)\} \subseteq T$, $\{p(a)\} \setminus T = \emptyset$, and $T \setminus \{p(a)\} = (T \setminus \{p(x)\}) \cup \{p(y) \mid y \neq a\}$, etc. Moreover, any literal/rule in a set is identified with its variants modulo variable renaming.

4.1 Normal form

Although extended abduction can handle the very general format of EDPs, some syntactic transformations are helpful. Based on [12] we will briefly describe how a semantically equivalent normal form of an abductive program $\langle K, \mathcal{A}\rangle$ is obtained – where both the program K and the set $\mathcal{A}$ of abducibles are EDPs. This makes an automatic handling of abductive programs easier; for example, abductive programs in normal form can be easily transformed into update programs as described in Section 4.2. The main step is that rules in $\mathcal{A}$ can be mapped to atoms by a naming function n. Let $\mathcal{R}$ be the set of abducible rules:

$$\mathcal{R} = \{\Sigma \leftarrow \Gamma \mid (\Sigma \leftarrow \Gamma) \in \mathcal{A} \text{ and } (\Sigma \leftarrow \Gamma) \text{ is not a literal}\}$$

Then the *normal form* $\langle K^n, \mathcal{A}^n\rangle$ is defined as follows where $n(R)$ maps each rule R to a fresh atom with the same free variables as R:

$$\begin{aligned} K^n &= (K \setminus \mathcal{R}) \cup \{\Sigma \leftarrow \Gamma, n(R) \mid R = (\Sigma \leftarrow \Gamma) \in \mathcal{R}\} \\ &\qquad \cup \{n(R) \mid R \in K \cap \mathcal{R}\} \\ \mathcal{A}^n &= (\mathcal{A} \setminus \mathcal{R}) \cup \{n(R) \mid R \in \mathcal{R}\} \end{aligned}$$

We define that any abducible literal L has the name L, i.e., $n(L) = L$. It is shown in [12], that for any observation O there is a 1-1 correspondence between (anti-)explanations with respect to $\langle K, A\rangle$ and those with respect to $\langle K^n, A^n\rangle$. That is, for $n(E) = \{n(R) | R \in E\}$ and $n(F) = \{n(R) | R \in F\}$: an observation O has a (minimal) skeptical (anti-)explanation (E, F) with respect to $\langle K, A\rangle$ iff O has a (minimal) skeptical (anti-)explanation $(n(E), n(F))$ with respect to $\langle K^n, A^n\rangle$. Hence, insertion (deletion) of a rule's name

in the normal form corresponds to insertion (deletion) of the rule in the original program. In sum, with the normal form transformation, any abductive program with abducible rules is reduced to an abductive program with only abducible literals.

Example 5. We transform the example knowledge base K into its normal form based on a set of abducibles that is identical to K: that is $\mathcal{A} = K$; a similar setting will be used in Section 5.2 to achieve deletion of formulas from K. Hence we transform $\langle K, \mathcal{A}\rangle$ into its normal form $\langle K^n, \mathcal{A}^n\rangle$ as follows where we write $n(R)$ for the naming atom of the only rule in $\mathcal{A}$:

$$\begin{aligned} K^n = \{&\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \quad \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \quad n(R), \\ &\mathit{Ill}(x, \mathsf{Aids}); \mathit{Ill}(x, \mathsf{Flu}) \leftarrow \mathit{Treat}(x, \mathsf{Medi1}), not\mathit{Treat}(x, \mathsf{Medi2}), n(R)\} \\ \mathcal{A}^n = \{&\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \quad \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \quad n(R)\ \} \end{aligned}$$

4.2 Update programs

Minimal (anti-)explanations can be computed with *update programs* (UPs) [12]. The *update-minimal* (U-minimal) answer sets of a UP describe which rules have to be deleted from the program, and which rules have to be inserted into the program, in order (un-)explain an observation.

For the given EDP K and a given set of abducibles $\mathcal{A}$, a set of **update rules** UR is devised that describe how entries of K can be changed. This is done with the following three types of rules.

1. **[Abducible rules]** The rules for abducible literals state that an abducible is either true in K or not. For each $L \in \mathcal{A}$, a new atom $\bar{L}$ is introduced that has the same variables as L. Then the set of abducible rules for each L is defined as
$$abd(L) := \{L \leftarrow not\bar{L}\ ,\ \bar{L} \leftarrow notL\}.$$
2. **[Insertion rules]** Abducible literals that are not contained in K might be inserted into K and hence might occur in the set E of the explanation (E, F). For each $L \in \mathcal{A} \setminus K$, a new atom $+L$ is introduced and the insertion rule is defined as
$$+L \leftarrow L.$$
3. **[Deletion rules]** Abducible literals that are contained in K might be deleted from K and hence might occur in the set F of the explanation (E, F). For each $L \in \mathcal{A} \cap K$, a new atom $-L$ is introduced and the deletion rule is defined as
$$-L \leftarrow notL.$$

The **update program** is then defined by replacing abducible literals in K with the update rules; that is,

$$UP = (K \setminus \mathcal{A}) \cup UR.$$

Example 6. Continuing Example 5, from $\langle K^n, \mathcal{A}^n\rangle$ we obtain

$$\begin{aligned} UP = \{\ &abd(\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})), \quad abd(\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})), \quad abd(n(R)), \\ &-\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}) \leftarrow not\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \\ &-\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}) \leftarrow not\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), \\ &-n(R) \leftarrow not\, n(R), \\ &\mathit{Ill}(x, \mathsf{Aids}); \mathit{Ill}(x, \mathsf{Flu}) \leftarrow \mathit{Treat}(x, \mathsf{Medi1}), not\mathit{Treat}(x, \mathsf{Medi2}), n(R)\} \end{aligned}$$

The set of atoms $+L$ is the set $\mathcal{UA}^+$ of positive update atoms; the set of atoms $-L$ is the set $\mathcal{UA}^-$ of negative update atoms. The set of **update atoms** is $\mathcal{UA} = \mathcal{UA}^+ \cup \mathcal{UA}^-$. From all answer sets of an update program UP we can identify those that are **update minimal** (U-minimal): they contain less update atoms than others. Thus, S is U-minimal iff there is no answer set T such that $T \cap \mathcal{UA} \subset S \cap \mathcal{UA}$.

4.3 Ground observations

It is shown in [9] how in some situations the observation formulas O can be mapped to new positive ground observations. Non-ground atoms with variables can be mapped to a new ground observation. Several positive observations can be conjoined and mapped to a new ground observation. A negative observation (for which an anti-explanation is sought) can be mapped as a NAF-literal to a new positive observation (for which then an explanation has to be found). Moreover, several negative observations can be mapped as a conjunction of NAF-literals to one new positive observation such that its resulting explanation acts as an anti-explanation for all negative observations together. Hence, in extended abduction it is usually assumed that O is a positive ground observation for which an explanation has to be found. In case of finding a skeptical explanation, an inconsistency check has to be made on the resulting knowledge base. Transformations to a ground observation and inconsistency check will be detailed in Section 5.1 and applied to confidentiality-preservation.

5 Confidentiality-Preservation with UPs

We now show how to achieve confidentiality-preservation by extended abduction: we define the set of abducibles and describe how a confidentiality-preserving knowledge base can be obtained by computing U-minimal answer sets of the appropriate update program. We additionally distinguish between the case that we allow only deletions of formulas – that is, in the anti-explanation (E, F) the set E of positive anti-explanation formulas is empty – and the case that we also allow insertions.

5.1 Policy transformation for credulous users

Elements of the confidentiality policy will be treated as negative observations for which an anti-explanation has to be found. Accordingly, we will transform policy elements to a set of rules containing new positive observations as sketched in Section 4.3. We will call these rules **policy transformation rules for credulous users** (PTR^{cred}).

More formally, assume $policy$ contains k elements. For each conjunction $C_i \in policy$ $(i = 1 \dots k)$, we introduce a new negative ground observation O_i^- and map C_i to O_i^-. As each C_i is a conjunction of (NAF-)literals, the resulting formula is an EDP rule. As a last policy transformation rule, we add one that maps all new negative ground observations O_i^- (in their NAF version) to a positive observation O^+. Hence,

$$PTR^{cred} := \{O_i^- \leftarrow C_i \mid C_i \in policy\} \cup \{O^+ \leftarrow not\, O_1^-, \dots, not\, O_k^-\}.$$

Example 7. The set of policy transformation rules for $policy'$ is

$$PTR^{cred} = \{O_1^- \leftarrow \mathit{Ill}(x, \mathsf{Aids})\ ,\ O_2^- \leftarrow \neg \mathit{AbleToWork}(x)\ ,\ O^+ \leftarrow not\, O_1^-, not\, O_2^-\}$$

Lastly, we consider a **goal rule** GR that enforces the single positive observation O^+: $GR = \{\leftarrow not\, O^+\}$.

We can also allow more expressive policy elements in disjunctive normal form (DNF: a disjunction of conjunctions of (NAF-)literals). If we map a DNF formula to a new observation (that is, $O_{disj}^- \leftarrow C_1 \vee \dots \vee C_l$) this is equivalent to mapping each conjunct to the observation (that is, $O_{disj}^- \leftarrow C_1, \dots, O_{disj}^- \leftarrow C_l$). We also semantically justify this splitting into disjuncts by arguing that in order to protect confidentiality of a disjunctive formula we indeed have to protect each disjunct alone. However, if variables are shared among disjuncts, these variables have to be grounded according to the Herbrand universe of $K \cup prior$ first; otherwise the shared semantics of these variables is lost.

5.2 Deletions for credulous users

As a simplified setting, we first of all assume that only deletions are allowed to achieve confidentiality-preservation. This setting can informally be described as follows: For a given knowledge base K, if we

only allow deletions of rules from K, we have to find a *skeptical negative explanation* F that explains the new positive observation O^+ while respecting $prior$ as invariable a priori knowledge. The set of abducibles is thus identical to K as we want to choose formulas from K for deletion: $\mathcal{A} = K$. That is, in total we consider the abductive program $\langle K, \mathcal{A}\rangle$. Then, we transform it into normal form $\langle K^n, \mathcal{A}^n\rangle$, and compute its update program UP as described in Section 4.2. As for $prior$, we add this set to the update program UP in order to make sure that the resulting answer sets of the update program do not contradict $prior$. Finally, we add all the policy transformation rules PTR^{cred} and the goal rule GR. The goal rule is then meant as a constraint that filters out those answer sets of $UP \cup prior \cup PTR^{cred}$ in which O^+ is *true*. We thus obtain a new program P as

$$P = UP \cup prior \cup PTR^{cred} \cup GR$$

and compute its U-minimal answer sets. If S is one of these answer sets, the negative explanation F is obtained from the negative update atoms contained in S: $F = \{L \mid -L \in S\}$.

To obtain a confidentiality-preserving knowledge base for a credulous user, we have to check for inconsistency with the negation of the positive observation O^+ (which makes F a *skeptical* explanation of O^+); and allow only answer sets of P that are U-minimal among those respecting this inconsistency property. More precisely, we check whether

$$(K \setminus F) \cup prior \cup PTR^{cred} \cup \{\leftarrow O^+\} \text{ is inconsistent.} \tag{1}$$

Example 8. We combine the update program UP of K with $prior$ and the policy transformation rules and goal rule. This leads to the following two U-minimal answer sets with only deletions which satisfy the inconsistency property (1):

$$S_1' = \{-\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), -\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), n(R), \overline{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})}, \overline{\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})}, O^+\}$$
$$S_2' = \{-\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), -n(R), \overline{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})}, \overline{n(R)}, O^+\}.$$

These answer sets correspond to the minimal solutions from Example 4 where $\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})$ must be deleted together with either $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})$ or the rule named R.

Theorem 1 (Correctness for deletions). *A knowledge base $K^{pub} = K \setminus F$ preserves confidentiality and changes K subset-minimally iff F is obtained by an answer set of the program P that is U-minimal among those satisfying the inconsistency property (1).*

Proof. (Sketch) First of all note that because we chose K to be the set of abducibles $\mathcal{A}$, only negative update atoms from $\mathcal{UA}^-$ occur in UP – no insertions with update atoms from $\mathcal{UA}^+$ will be possible. Hence we automatically obtain an anti-explanation (E, F) where E is empty. As shown in [12], there is a 1-1 correspondence of minimal explanations and U-minimal answer sets of update programs; and anti-explanations are identical to explanations of a new positive observation when applying the transformations as in PTR^{cred}. By properties of skeptical (anti-)explanations we have thus $K^{pub} \cup prior \cup PTR^{cred} \models O^+$ but for every O_i^- there is no answer set in which O_i^- is satisfied. This holds iff for every policy element C_i there is no answer set of $K^{pub} \cup prior$ that satisfies any instantiation of C_i (with respect to the Herbrand universe of $K^{pub} \cup prior$); thus $cred(K^{pub} \cup prior, C_i) = \emptyset$. Subset-minimal change carries over from U-minimality of answer sets.

5.3 Deletions and literal insertions

To obtain a confidentiality-preserving knowledge base, (incorrect) entries may also be inserted into the knowledge base. To allow for insertions of literals, a more complex set $\mathcal{A}$ of abducibles has to be chosen. We reinforce the point that the subset $\mathcal{A} \cap K$ of abducibles that are already contained in the knowledge base K are those that may be deleted while the subset $\mathcal{A} \setminus K$ of those abducibles that are not contained in K may be inserted.

First of all, we assume that the policy transformation is applied as described in Section 5.1. Then, starting from the new negative observations O_i^- used in the policy transformation rules, we trace back all rules in $K \cup prior \cup PTR^{cred}$ that influence these new observations and collect all literals in the bodies

of these rules. In other words, we construct a dependency graph (as in [16]) and collect the literals that the negative observations depend on. More formally, let P_0 be the set of literals that the new observations O_i^- directly depend on:

$$P_0 = \{L \mid L \in body(R) \text{ or } notL \in body(R) \\ \text{where } R \in PTR^{cred} \text{ and } O_i^- \in head(R)\}$$

Next we iterate and collect all the literals that the P_0 literals depend on:

$$P_{j+1} = \{L \mid L \in body(R) \text{ or } notL \in body(R) \\ \text{where } R \in K \cup prior \cup PTR^{cred} \text{ and } head(R) \cap P_j \neq \emptyset\}$$

and combine all such literals in a set $\mathcal{P} = \bigcup_{j=0}^{\infty} P_j$.

As we also want to have the option to delete rules from K (not only the literals in $\mathcal{P}$), we define the set of abducibles as the set $\mathcal{P}$ plus all those rules in K whose head depends on literals in $\mathcal{P}$:

$$\mathcal{A} = \mathcal{P} \cup \{R \mid R \in K \text{ and } head(R) \cap \mathcal{P} \neq \emptyset\}$$

Example 9. For the example $K \cup prior \cup PTR^{cred}$, the dependency graph is shown in Figure 2. We note that the new negative observation O_1^- directly depends on the literal $\mathit{Ill}(x, \mathsf{Aids})$ and the new negative observation O_2^- directly depends on the literal $\neg\mathit{AbleToWork}(x)$; this is the first set of literals $P_0 = \{\mathit{Ill}(x, \mathsf{Aids}), \neg\mathit{AbleToWork}(x)\}$. By tracing back the dependencies in the graph,

$$\mathcal{P} = \{\mathit{Ill}(x, \mathsf{Aids}), \neg\mathit{AbleToWork}(x), \mathit{Ill}(x, \mathsf{Flu}), \mathit{Treat}(x, \mathsf{Medi1}), \mathit{Treat}(x, \mathsf{Medi2})\}$$

is obtained. Lastly, we also have to add the rule R from K to $\mathcal{A}$ because literals in its head are contained in $\mathcal{P}$.

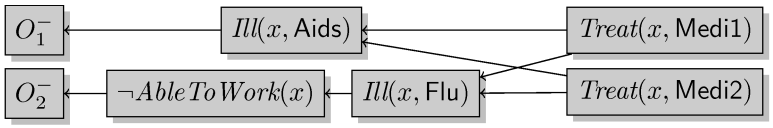


Fig. 2. Dependency graph for literals in $K \cup prior \cup PTR$

We obtain the normal form and then the update program UP for K and the new set of abducibles $\mathcal{A}$. The process of finding a skeptical explanation proceeds with finding an answer set of program P as in Section 5.2 where additionally the positive explanation E is obtained as $E = \{L \mid +L \in S\}$ and S is U-minimal among those satisfying

$$(K \setminus F) \cup E \cup prior \cup PTR^{cred} \cup \{\leftarrow O^+\} \text{ is inconsistent.} \tag{2}$$

Example 10. For UP from Example 8 the new set of abducibles leads to additional insertion rules. Among others, the insertion rule for the new abducible $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2})$ is $+\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2}) \leftarrow \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2})$. With this new rule included in UP, we also obtain the solution of Example 4 where the fact $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2})$ is inserted into K (together with deletion of $\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})$).

Theorem 2 (Correctness for deletions & literal insertions). *A knowledge base $K^{pub} = (K \setminus F) \cup E$ preserves confidentiality and changes K subset-minimally iff (E, F) is obtained by an answer set of program P that is U-minimal among those satisfying inconsistency property (2).*

Proof. (Sketch) In UP, positive update atoms from $\mathcal{UA}^+$ occur for literals on which the negative observations depend. For subset-minimal change, only these literals are relevant for insertions; inserting other literals will lead to non-minimal change. In analogy to Theorem 1, by the properties of minimal skeptical (anti-)explanations that correspond to U-minimal answer sets of an update program, we obtain a confidentiality-preserving K^{pub} with minimal change.

6 Discussion and Conclusion

This article showed that when publishing a logic program, confidentiality-preservation can be ensured by extended abduction; more precisely, we showed that under the credulous query response it reduces to finding skeptical anti-explanations with update programs. This is an application of data modification, because a user can be mislead by the published knowledge base to believe incorrect information; we hence apply dishonesties [11] as a security mechanism. This is in contrast to [16] whose aim is to avoid incorrect deductions while enforcing access control on a knowledge base. Another difference to [16] is that they do not allow disjunctions in rule heads; hence, to the best of our knowledge this article is the first one to handle a confidentiality problem for EDPs. In [3] the authors study databases that may provide users with incorrect answers to preserve security in a multi-user environment. Different from our approach, they consider a database as a set of formulas of propositional logic and formulate the problem using modal logic. In analogy to [12], a complexity analysis for our approach can be achieved by reduction of extended abduction to normal abduction. Work in progress covers data publishing for skeptical users; future work might handle insertion of non-literal rules.

References

1. Foto N. Afrati and Phokion G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT2009*, volume 361 of *ACM International Conference Proceeding Series*, pages 31–41. ACM, 2009.
2. Joachim Biskup. Usability confinement of server reactions: Maintaining inference-proof client views by controlled interaction execution. In *DNIS 2010*, volume 5999 of *LNCS*, pages 80–106. Springer, 2010.
3. Piero A. Bonatti, Sarit Kraus, and V. S. Subrahmanian. Foundations of secure deductive databases. *IEEE Trans. Knowl. Data Eng.*, 7(3):406–422, 1995.
4. Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *LPNMR 2011*, volume 6645 of *LNCS*, pages 388–403. Springer, 2011.
5. Jürgen Dix, Wolfgang Faber, and V. S. Subrahmanian. The relationship between reasoning about privacy and default logics. In *LPAR 2005*, volume 3835 of *Lecture Notes in Computer Science*, pages 637–650. Springer, 2005.
6. Csilla Farkas and Sushil Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–11, 2002.
7. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.
8. Bernardo Cuenca Grau and Ian Horrocks. Privacy-preserving query answering in logic-based information systems. In *ECAI2008*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 40–44. IOS Press, 2008.
9. Katsumi Inoue and Chiaki Sakama. Abductive framework for nonmonotonic theory change. In *Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, volume 1, pages 204–210. Morgan Kaufmann, 1995.
10. Jiefei Ma, Alessandra Russo, Krysia Broda, and Emil Lupu. Multi-agent confidential abductive reasoning. In *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 175–186. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
11. Chiaki Sakama. Dishonest reasoning by abduction. In *22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 1063–1064. IJCAI/AAAI, 2011.
12. Chiaki Sakama and Katsumi Inoue. An abductive framework for computing knowledge base updates. *Theory and Practice of Logic Programming*, 3(6):671–713, 2003.
13. Phiniki Stouppa and Thomas Studer. Data privacy for knowledge bases. In Sergei N. Artëmov and Anil Nerode, editors, *LFCS2009*, volume 5407 of *LNCS*, pages 409–421. Springer, 2009.
14. Tyrone S. Toland, Csilla Farkas, and Caroline M. Eastman. The inference problem: Maintaining maximal availability in the presence of database updates. *Computers & Security*, 29(1):88–103, 2010.
15. Lena Wiese. Horizontal fragmentation for data outsourcing with formula-based confidentiality constraints. In *IWSEC 2010*, volume 6434 of *LNCS*, pages 101–116. Springer, 2010.
16. Lingzhong Zhao, Junyan Qian, Liang Chang, and Guoyong Cai. Using ASP for knowledge management with user authorization. *Data & Knowl. Eng.*, 69(8):737–762, 2010.

The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. We report about the current state and designated features of the tool SeaLion, aimed to serve as an integrated development environment (IDE) for answer-set programming (ASP). A main goal of SeaLion is to provide a user-friendly environment for supporting a developer to write, evaluate, debug, and test answer-set programs. To this end, new support techniques have to be developed that suit the requirements of the answer-set semantics and meet the constraints of practical applicability. In this respect, SeaLion benefits from the research results of a project on methods and methodologies for answer-set program development in whose context SeaLion is realised. Currently, the tool provides source-code editors for the languages of Gringo and DLV that offer syntax highlighting, syntax checking, and a visual program outline. Further implemented features are support for external solvers and visualisation as well as visual editing of answer sets. SeaLion comes as a plugin of the popular Eclipse platform and provides itself interfaces for future extensions of the IDE.

1 Introduction

Answer-set programming (ASP) is a well-known and fully declarative problem-solving paradigm based on the idea that solutions to computational problems are represented in terms of logic programs such that the models of the latter, referred to as the *answer sets*, provide the solutions of a problem instance.[1] In recent years, the expressibility of languages supported by answer-set solvers increased significantly [3]. As well, ASP solvers have become much more efficient, e.g., the solver Clasp proved to be competitive with state-of-the-art SAT solvers [4].

Despite these improvements in solver technology, a lack of suitable *engineering tools* for developing programs is still a handicap for ASP towards gaining widespread popularity as a problem-solving paradigm. This issue is clearly recognised in the ASP community and work to fill this gap has started recently, addressing issues like debugging, testing, and the modularity of programs [5–13]. Additionally, in order to facilitate tool support as known for other programming languages, attempts to provide *integrated development environments* (IDEs) have been put forth. Work in this direction includes the systems APE [14], ASPIDE [15], and iGROM [16].

Following this endeavour, in this paper, we describe the current status and designated features of a further IDE, SeaLion, developed as part of an ongoing research project on methods and methodologies for developing answer-set programs [17].

SeaLion is designed as an Eclipse plugin, providing useful and intuitive features for ASP. Besides experts, the target audience for SeaLion are software developers new to ASP, yet who are familiar with support tools as used in procedural and object-oriented programming. Our goal is to fully support the languages of the current state-of-the-art solvers Clasp (in conjunction with Gringo) [3, 18] and DLV [19], which distinguishes SeaLion from the other IDEs mentioned above which support only a single solver. Indeed, APE [14], which is also an Eclipse plugin, supports only the language of Lparse [20] that is a subset of the language of Gringo, whilst ASPIDE [15], a recently developed standalone IDE, offers support only for DLV programs. Although iGROM provides basic functionality for the languages of both Lparse and DLV [16], it currently does not support the latest version of DLV or the full syntax of Gringo.

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

[1] For an overview about ASP, we refer the reader to a survey article by Gelfond and Leone [1] or the textbook by Baral [2].

At present, SeaLion is in an alpha version that already implements important core functionality. In particular, the languages of DLV and Gringo are supported to a large extent. The individual parsers translate programs and answer sets to data structures that are part of a rich and flexible framework for internally representing program elements. Based on these structures, the editor provides syntax highlighting, syntax checks, error reporting, error highlighting, and automatic generation of a program outline. There is functionality to manage external tools such as answer-set solvers and to define arbitrary pipes between them (as needed when using separate grounders and solvers). Moreover, in order to run an answer-set solver on the created programs, launch configurations can be created in which the user can choose input files, a solver configuration, command line arguments for the solver, as well as output-processing strategies. Answer sets resulting from a launch can either be parsed and stored in a view for interpretations, or the solver output can be displayed unmodified in Eclipse's built-in console view.

Another key feature of SeaLion is the capability for the *visualisation* and *visual editing* of interpretations. This follows ideas from the visualisation tools ASPVIZ [21] and IDPDraw [22], where a visualisation program Π_V (itself being an answer-set program) is joined with an interpretation I that shall be visualised. Subsequently, the overall program is evaluated using an answer-set solver, and the visualisation is generated from a resulting answer set. However, the editing feature of SeaLion allows also to graphically manipulate the interpretations under consideration which is not supported by ASPVIZ and IDPDraw.

The visualisation functionality of SeaLion is itself represented as an Eclipse plugin, called Kara.[2] In this paper, however, we describe only the basic functionality of Kara; a full description is given in a companion paper [23].

2 Architecture and Implementation Principles

We assume familiarity with the basic concepts of answer-set programming (ASP) (for a thorough introduction to the subject, cf. Baral [2]). In brief, an answer-set program consists of rules of the form

$$a_1 \vee \cdots \vee a_l :- a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, and all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol $\neg$). For a rule r as above, the expression left to the symbol "$:-$" is the *head* of r and the expression to the right of "$:-$" is the *body* of r. If $n = l = 1$, r is a *fact*; if r contains no disjunction, r is *normal*; and if $l = 0$ and $n > 0$, r is a *constraint*. For facts, the symbol "$:-$" is usually omitted. The *grounding* of a program P relative to its Herbrand universe is defined as usual. An *interpretation* I is a finite and consistent set of ground literals, where consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a. I is an *answer set* of a program P if it is a minimal model of the grounding of the *reduct* of P relative to I (see Baral [2] for details).

A key aspect in the design of SeaLion is extensibility. That is, on the one hand, we want to have enough flexibility to handle further ASP languages such that previous features can deal with them with no or little adaption. On the other hand, we want to provide a powerful API framework that can be used by future features. To this end, we defined a hierarchy of classes and interfaces that represent *program elements*, i.e., fragments of ASP languages. This is done in a way such that we can use common interfaces and base classes for representing similar program elements of different ASP languages. For instance, we have different classes for representing literals of the Gringo language and literals of the DLV language in order to be able to handle subtle differences. For example, in Gringo, a literal can have several other literals as conditions, e.g., `redEdge(X,Y):edge(X,Y):red(X):red(Y)`. Intuitively, during grounding, this literal is replaced by the list of all literals `redEdge(n1,n2)`, where `edge(n1,n2)`, `red(n1)`, and `red(n2)` can be derived during grounding. As DLV is unaware of conditions, an object of class DLVStandardLiteral has no support for them, whereas a GringoStandardLiteral object keeps a list of condition literals. Substantial differences in other language features, like aggregates,

[2] The name derives, with all due respect, from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

optimisation, and filtering support, are also reflected by different classes for Gringo and DLV, respectively. However, whenever possible, these classes are derived from a common base class or share common interfaces. Therefore, plugins can, for example, use a general interface for aggregate literals to refer to aggregates of both languages. Hence, current and future feature implementations can make use of high-level interfaces and stay independent of the concrete ASP language to a large extent.

Also, within the SeaLion implementation, the aim is to have independent modules for different features, in form of Eclipse plugins, that ensure a well-structured code. Currently, there are the following plugins: (i) the main plugin, (ii) a plugin that adapts the ANTLR parsing framework [24] to our needs, (iii) two solver plugins, one for Gringo/Clasp and one for DLV, and (iv) the Kara plugin for answer-set visualisation and visual editing. Moreover, it is a key aim to smoothly integrate SeaLion in the Eclipse platform and to make use of functionality the latter provides wherever suitable. The motivation is to exploit the rich platform as well as to ensure compatibility with upcoming versions of Eclipse.

The decision to build on Eclipse, rather than writing a stand-alone application from scratch, has many benefits. For one, we profit from software reuse as we can make use of the general GUI of Eclipse and just have to adapt existing functionality to our needs. Examples include the text editor framework, source-code annotations, problem reporting and quick fixes, project management, the undo-redo mechanism, the console view, the navigation framework (Outline, Project Explorer), and launch configurations. Moreover, much functionality of Eclipse can be used without any adaptions, e.g., workspace management, the possibility to define working sets, i.e., grouping arbitrary files and resources together, software versioning and revision control (e.g., based on SVN or CVS), and task management. Another clear benefit is the popularity of Eclipse among software developers, as it is a widely used standard tool for developing Java applications. Arguably, people who are familiar with Eclipse and basic ASP skills will easily adapt to SeaLion. Finally, choosing Eclipse for an IDE for ASP offers a chance for integration of development tools for hybrid languages, i.e., combinations of ASP and procedural languages. For instance, Gringo supports the use of functions written in the LUA scripting language [25]. As there is a LUA plugin for Eclipse available, one can at least use that in parallel with SeaLion, however there is also potential for a tighter integration of the two plugins.

The sources of SeaLion are available for download from

`http://sourceforge.net/projects/mmdasp/.`

An Eclipse update site will be made available as soon as SeaLion reaches beta status.

3 Current Features

In this section, we describe the features that are already operational in SeaLion, including technical details on the implementation.

3.1 Source-Code Editor

The central element in SeaLion is the *source-code editor* for logic programs. For now, it comes in two variations, one for DLV and one for Gringo. A screenshot of a Gringo source file in SeaLion's editor is given in Fig. 1. By default, files with names ending in ".lp", ".lparse", ".gr", or ".gringo" are opened in the Gringo editor, whereas files with extensions ".dlv" or ".dl" are opened in the DLV editor. Nevertheless, any file can be opened in either editor if required.

The editors provide *syntax highlighting*, which is computed in two phases. Initially, a fast syntactic check provides initial colouring and styling for comments and common tokens like dots concluding rules and the rule implication symbol. While editing the source code, after a few moments of user inactivity, the source code is parsed and data structures representing the program are computed and stored for various purposes. The second phase of syntax highlighting is already based on this program representation and allows for fine-grained highlighting depending not only on the type of the program element but also on its role. For instance, a literal that is used in the condition of another literal is highlighted in a different way than stand-alone literals.

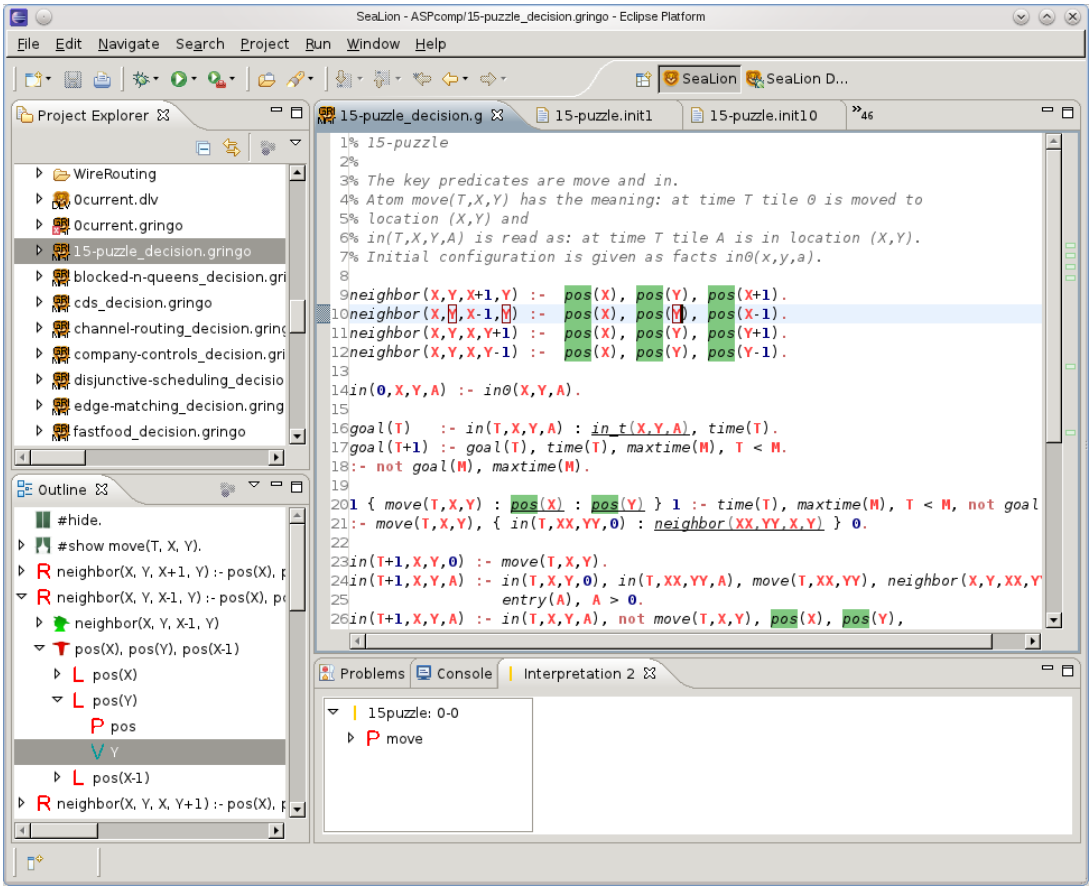


Fig. 1. A screenshot of SeaLion's editor, the program outline, and the interpretation view.

The parsers used are based on the ANTLR framework [24] and are in some respect *more lenient than the respective solver parsers*. For one thing, they are more tolerant towards syntax errors. For instance, in many cases they accept terms of various types (constants, variables, aggregate terms) where a solver requires a particular type, like a variable. The errors will still be noticed, during building the program representation or afterwards, by means of explicit checks. This tolerance allows for more specific warning and error reporting than provided by the solvers. For example, the system can warn the user that he or she used a constant on the left-hand side of an assignment where only a variable is allowed. Another parsing difference is the *handling of comments*. The parser does not throw them away but collects them and associates them to the program elements in their immediate neighbourhood. One benefit is that the information contained in comments can be kept when performing automatic transformations on the program, like rule reorderings or translations to other logic programming dialects. Another advantage is that we can make use of comments for enriching the language with our own *meta-statements* that do not interfere with the solver when running the file. We reserved the token "%!" for initiating meta commands and "%*!" and "*%" for the start and end of block meta commands in the Gringo editor, respectively. Currently, one type of meta command is supported: assigning properties to program elements.

Example 1. In the following source code, a meta statement assigns the name "r1" to the rule it precedes.

```
%! name = r1;
a(X) :- c(X).
```

These names are currently used in a side application of SeaLion for reifying disjunctive non-ground programs as used in a previous debugging approach [10]. Moreover, names assigned to program elements as above can be seen in Eclipse's "Outline View". SeaLion uses this view to give an overview of the edited

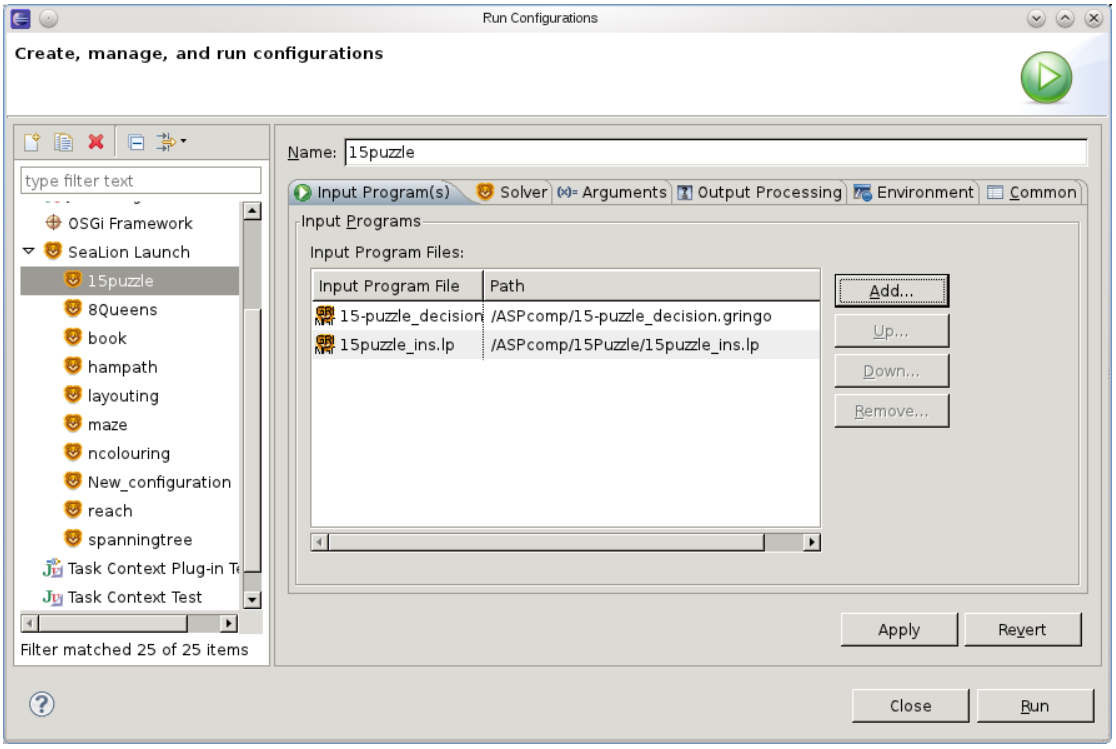


Fig. 2. Selecting two source files for ASP solving in Eclipse's launch configuration dialog.

program in a tree-shaped graphical representation. The rules of the programs are represented by the nodes of depth 1 of this tree. By expanding the ancestor nodes of an individual rule, one can see its elements, i.e., head, body, literals, predicates, terms, etc. Clicking on such an element selects the corresponding program code in the editor, and the programmer can proceed editing there. A similar outline is also available in Eclipse's "Project Explorer", as subtree under the program's source file.

Another feature of the editor is the support for *annotations*. These are means to temporarily highlight parts of the source code. For instance, SeaLion annotates occurrences of the program element under the text cursor. If the cursor is positioned over a literal, all literals of the same predicate are highlighted in the text as well as in a bar next to the vertical scrollbar that indicates the positions of all occurrences in the overall document. Likewise, when a constant or a variable in a rule is on the cursor position, their occurrences are detected within the whole source code or within the rule, respectively.

Another application of annotations is *problem reporting*. Syntax errors and warnings are displayed in two ways. First, as annotations in the source code, they are marked with a zig-zag styled underline. Second, they are displayed in Eclipse's "Problem View" that collects various kinds of problems and allows for directly jumping to the problematic source code region upon mouse click.

3.2 Support for External Tools

In order to interact with solvers and grounders from SeaLion, we implemented a mechanism for handling external tools. One can define *external tool configurations* that specify the path to an executable as well as default command-line parameters. Arbitrary command-line tools are supported; however, there are special configuration types for some programs such as Gringo, Clasp, and DLV. For these, it is planned to have a specialised GUI that allows for a more convenient modification of command-line parameters. In addition to external command-line tools, one can also define tool configurations that represent pipes between external tools. This is needed when grounding and solving are provided by separate executables. For instance, one can define two separate tool configurations for Gringo and Clasp and define a piped tool configuration

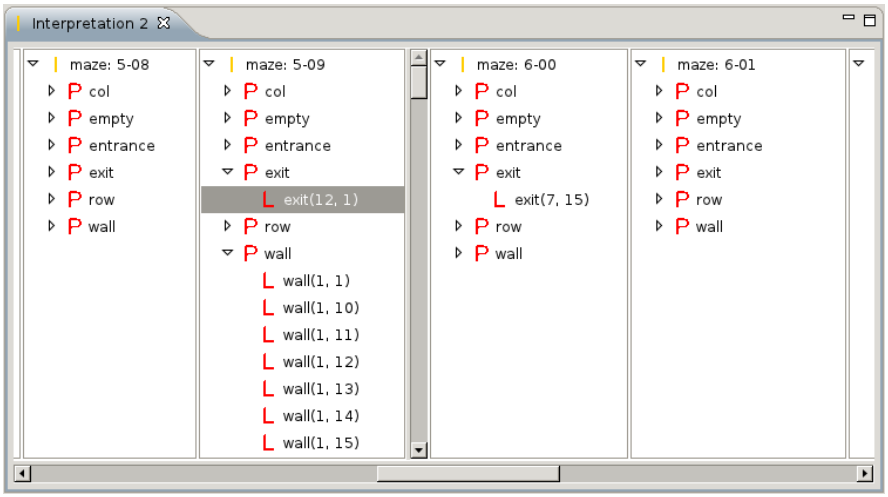


Fig. 3. SeaLion's interpretation view.

for using the two tools in a pipe. Pipes of arbitrary length are supported such that arbitrary pre- and post-processing can be done when needed.

For executing answer-set solvers, we make use of Eclipse's "launch configuration framework". In our setting, a launch configuration defines which programs should be executed using which solver. Figure 2 shows the the page of the launch configuration editor on which input files for a solver invocation can be selected.

Besides using the standard command-line parameters from the tool configurations, also customised parameters can be set for the individual program launches.

3.3 Interpretation View

The programmer can define how the output of an ASP solver run should be treated. One option is to print the solver output as it is for Eclipse's "console view". The other option is to parse the resulting answer sets and store them in SeaLion's *interpretation view* that is depicted in Fig. 3. Here, interpretations are visualised as expandable trees of depth 3. The root node is the interpretation (marked by a yellow "I"), and its children are the predicates (marked by a red "P") appearing in the interpretation. Finally, each of these predicates is the parent node of the literals over the predicate that are contained in the interpretation (marked by a red "L"). Compared to a standard textual representation, this way of visualising answer sets provides a well-arranged overview of the individual interpretations. We find it also more appealing than a tabular representation where only entries for a single predicate are visible at once. Moreover, by horizontally arranging trees for different interpretations next to each other, it is easy to compare two or more interpretations.

The interpretation view is not only meant to provide a good visualisation of results, but also serves as a starting point for ASP developing tools that depend on interpretations. One convenient feature is dragging interpretations or individual literals from the interpretation view and dropping them on the source-code editor. When released, these are transformed into facts of the respective ASP language.

3.4 Visualisation and Visual Editing

The plugin Kara [23] is a tool for the graphical visualisation and editing of interpretations. It is started from the interpretation view. One can select an interpretation for visualisation by right-clicking it in the view and choose between a *generic visualisation* or a *customised visualisation*. The latter is specified by the user by means of a visualisation answer-set program. The former represents the interpretation as a labelled hypergraph.

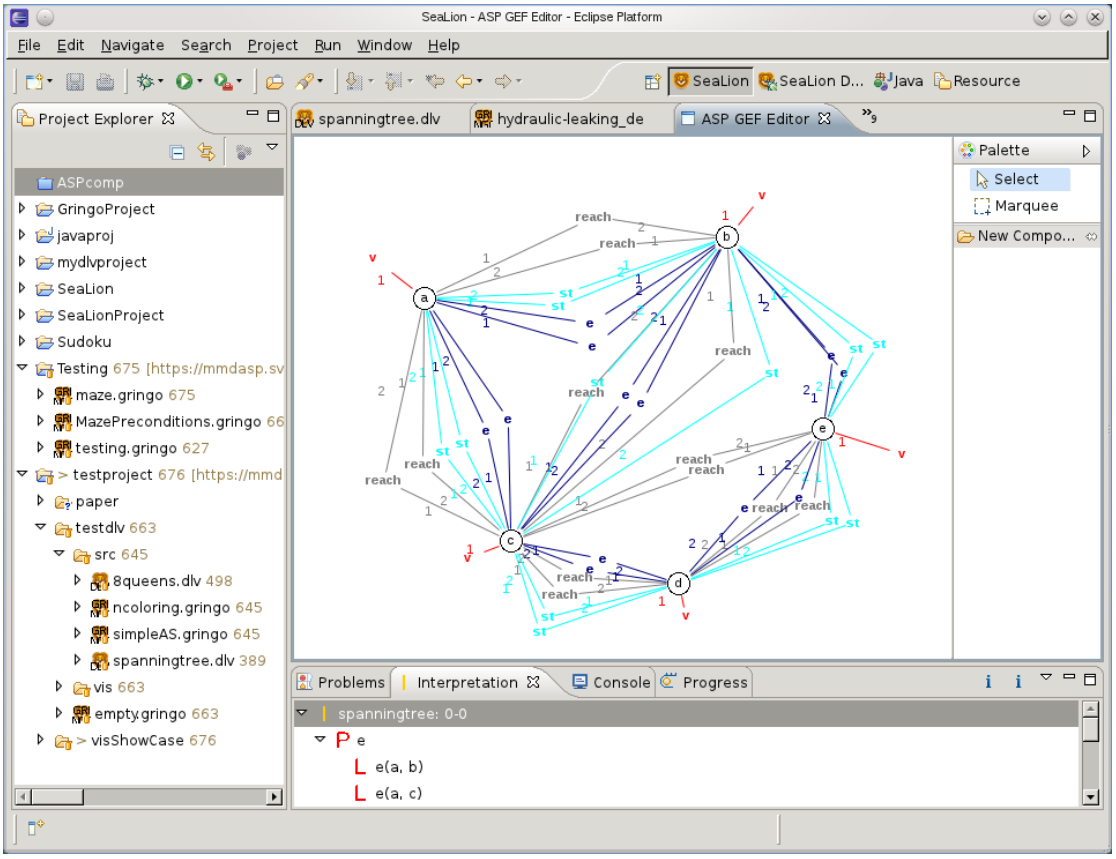


Fig. 4. A screenshot of SeaLion's visual interpretation editor.

In the generic visualisation, the nodes of the hypergraph are the individuals appearing in the interpretation. The edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of an edge are used for expressing the argument position of the individual. In order to distinguish between different predicates, each edge has an additional label stating the predicate name. Moreover, edges of the same predicate are of the same colour. An example of a generic visualisation of a spanning tree interpretation is shown in Fig. 4 (the layout of the graph has been manually optimised in the editor).

The customised visualisation feature allows for specifying how the interpretation should be illustrated by means of an answer-set program that uses a powerful pre-defined visualisation vocabulary. The approach follows the ideas of ASPVIZ [21] and IDPDraw [22]: a visualisation program Π_V is joined with the interpretation I to be visualised (technically, I is considered as a set of facts) and evaluated using an answer-set solver. One of the resulting answer sets, I_V, is then interpreted by SeaLion for building the graphical representation of I. The vocabulary allows for using and positioning basic graphical elements such as lines, rectangles, polygons, labels, and images, as well as graphs and grids composed of such elements.

The resulting visual representation of an interpretation is shown in a graphical editor that also allows for manipulating the visualisation in many ways. Properties such as colours, IDs, and labels can be manipulated and graphical elements can be repositioned, deleted, or even created. This is useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG) for use outside of SeaLion, using our SVG export functionality. Second, modifying the visualisation can be used to obtain a modified version I' of the visualised interpretation I by abductive reasoning.

In fact, we implemented a feature that allows for abducing an interpretation that would result in the modified visualisation. Modifications in the visual editor are automatically reflected in an adapted version

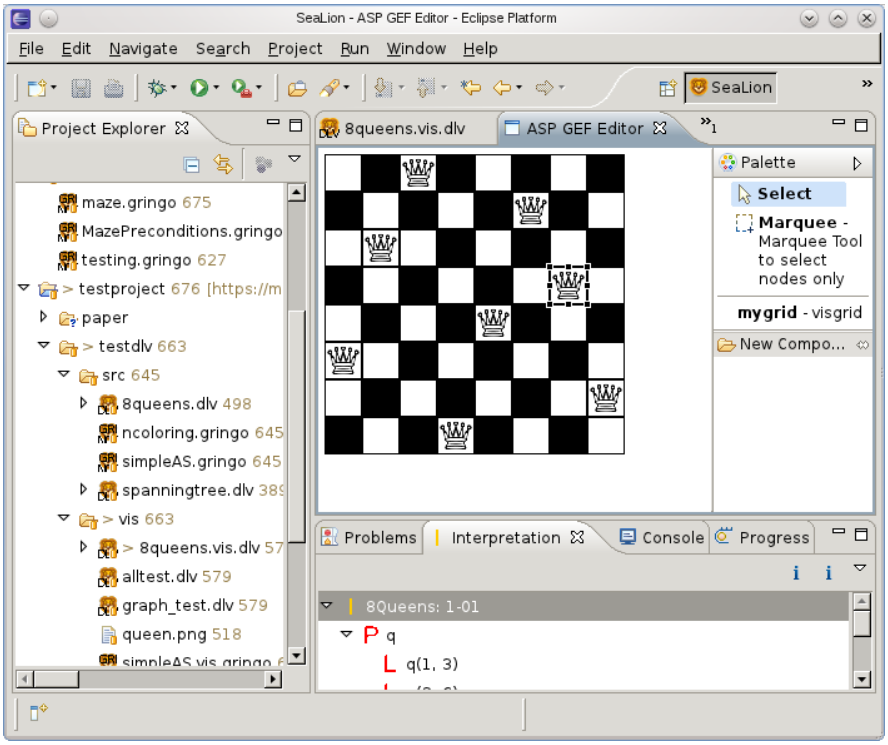


Fig. 5. A customised visualisation of an 8-queens instance.

I'_V of the answer set I_V representing the visualisation. We then use an answer-set program $\lambda(I'_V, \Pi_V)$ that is constructed depending on the modified visualisation answer set I'_V and the visualisation program Π_V for obtaining the modified interpretation I' as a projected answer set of $\lambda(I'_V, \Pi_V)$. For more details, we refer to a companion paper [23]. An example for a customised visualisation for a solution to the 8-queens problem is given in Fig. 5.

4 Projected Features

In the following, we give an overview of further functionality that we plan to incorporate into `SeaLion` in the near future.

One core feature that is already under development is the support for *stepping-based debugging* of answer-set programs as introduced in recent work [26]. Here, we aim for an intuitive and easy-to-handle user interface, which is clearly a challenge to achieve for reasons intrinsic to ASP. In particular, the discrepancy of having non-ground programs but solutions based on their groundings makes the realisation of practical debugging tools for ASP non-trivial.

We want to enrich `SeaLion` with support for *typed predicates*. That is, the user can define the domain for a predicate. For instance consider a predicate `age/2` stating the age of a person. Then, with typing, we can express that for every atom `age(t1,t2)`, the term `t1` represents an element from a set of persons, whereas `t2` represents an integer value. Two types of domain specifications will be supported, namely direct ones, which explicitly state the names of the individuals of the domain, and indirect ones that allow for specifications in terms of the domain of other predicates. We expect multiple benefits from having this kind of information available. First, it is useful as a documentation of the source code. A programmer can clearly specify the intended meaning of a predicate and look it up in the type specifications. Moreover, type violations in the source code of the program can be automatically detected as illustrated by the following example.

Example 2. Assume we want to define a rule deriving atoms with predicate symbol `serves/3`, where `serves(R,D,P)` expresses that restaurant `R` serves dish `D` at price `P`. Furthermore, the two predicates `dishAvailable/2` and `price/3` state which dishes are currently available in which restaurants and the price of a dish in a restaurant, respectively. Assume we have type specifications stating that for `serves(R,D,P)` and `dishAvailable(D,R)`, `R` is of type `restaurant` and `D` of type `dish`. Then, a potential type violation in the rule

```
serves(R,D,P) :- dishAvailable(R,D),price(R,D,P)
```

could be detected, where the programmer mixed up the order of variables in `dishAvailable(R,D)`.

In order to avoid problems like in the above example in the first place, autocompletion functionality could be implemented such that variables and constants of correct types are suggested when writing the arguments of a literal in a rule. Technically, we plan to realise type definitions within program comments, similar to other meta-statements as sketched in Section 3.

We want to combine the typing system with functionality that allows for defining *program signatures*. One application of such signatures is for specifying the predicates and terms used for abducing a modified interpretation I' in our plugin for graphically editing interpretations. Moreover, input and output signatures can be defined for uniform problem encodings, i.e., answer-set programs that expect a set of facts representing a problem instance as input such that its answer sets correspond to the solutions for this instance. Then, such signatures can be used in our planned support for *assertions* that will allow for defining pre- and post-conditions of answer-set programs. Having a full specification for the input of a program, i.e., a typed signature and input constraints in the form of preconditions, one can automatically generate input instances for the program and use them, e.g., for random testing [12]. Also, more advanced testing and verification functionality can be realised, like the automatic generation of valid input (with respect to the pre-conditions) that violates a post-condition.

In order to reduce the amount of time a programmer has to spend for writing type and signature definitions, we want to explore methods for partially extracting them from the source code or from interpretations.

Other projected features include typical amenities of Eclipse editors such as refactoring, autocompletion, pretty-printing, and providing quick-fixes for typical problems in the source code. Moreover, checks for errors and warnings that are not already detected by the parser, for example for detecting unsafe variables, need still to be implemented.

We also want to provide different kinds of program translations in `SeaLion`. To this end, we already implemented a flexible framework for transforming program elements to string representations following different strategies. In particular, we aim at translations between different solver languages at the non-ground level. Here, we first have to investigate strategies when and how transformations of, e.g., aggregates can be applied such that a corresponding overall semantics can be achieved. Other specific program translations that we consider for implementation would be necessary for realising the import and export of rules in the Rule Interchange Format (RIF) [27] which is a W3C recommendation for exchanging rules in the context of the Semantic Web. Notably, a RIF dialect for answer-set programming, called RIF-CASPD, has been proposed [28].

Further convenience improvements regarding the use of external tools in `SeaLion` include the support for setting default solvers for different languages and a specialised GUI for choosing the command-line parameters. For launch configurations, we want to add the possibility to directly write the output of a tool invocation into a file and to allow for exporting the launch configuration as native stand-alone scripts.

Finally, there are many possible ways to enhance the GUI of `SeaLion`. We want to extend the support for drag-and-drop operations such that, e.g., program elements in the outline can be dragged into the editor. Moreover, we plan to realise sorting and filtering features for the outline and interpretation view. Regarding interpretations, we aim for supporting textual editing of interpretations directly in the view, besides visual editing, and a feature for comparing multiple interpretations by highlighting their differences.

5 Related Work

In this section, we give a short overview of existing IDEs for core ASP languages. To begin with, the tool `APE` that has been developed at the University of Bath [14] is also based on Eclipse. It supports

the language of Lparse and provides syntax highlighting, syntax checking, program outline, and launch configuration. Additionally, APE has a feature to display the predicate dependency graph of a program. ASPIDE, a recent IDE for DLV programs [15], is a standalone tool that already offers many features as it builds on previous tools [29–31]. Some functionality we want to incorporate in SeaLion is already supported by ASPIDE, e.g., code completion, refactoring, and quick fixes. Further features of ASPIDE are support for code templates and a visual program editor. We do not aim for comprehensive visual source-code editing in SeaLion but consider the use of program templates that allow for expressing common programming patterns. One disadvantage of ASPIDE is that the tracing component of the IDE [30] is not publicly available. In their current releases, neither APE nor ASPIDE support graphical visualisation or visual editing of answer sets as available in SeaLion. ASPIDE allows for displaying answer sets in a tabular form. This is an improvement compared to the standard textual representation but comes with the drawback that only entries for a single predicate are visible at once. Besides the graphical representation, SeaLion can display interpretations in a dedicated view that gives a good overview of the individual interpretations and allows also to compare different interpretations.

Concerning supported ASP languages, SeaLion is the first IDE to support the language of Gringo, rather than its Lparse subset. Moreover, other proposed IDEs for ASP do only consider the language of either DLV or Lparse, with the exception of iGROM that provides basic syntax highlighting and syntax checking for the languages of both, Lparse and DLV [16]. Note that iGROM has been developed at our department independently from SeaLion as a student project. A speciality of iGROM is the support for the front-end languages for planning and diagnosis of DLV. There also exist proprietary IDEs for ASP related languages with support for object-oriented features, OntoStudio and OntoDLV [32, 33].

Compared to ASPVIZ [21] and IDPDraw [22], our plugin Kara [23] allows not only for visualisation of an interpretation but also for visually editing the graphical representation such that changes are reflected in the visualised interpretation. Moreover, Kara offers support for generic visualisation, automatic layout of graph structures, and special support for grids.

6 Conclusion

In this paper, we presented the current status of SeaLion, an IDE for ASP languages that is currently under development. We discussed general principles that we follow in our implementation and gave an overview of current and planned features. SeaLion is an Eclipse plugin and supports the ASP languages of Gringo and DLV. The most important step in the advancement of the IDE is the integration of an easy-to-use debugging system.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - The A-Prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England, UK (2003)
3. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver clasp: Progress report. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009). Volume 5753 of Lecture Notes in Computer Science, Springer (2009) 509–514
4. SAT 2009 competition: http://www.satcompetition.org/2009/
5. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proceedings of the 3rd Workshop on Answer Set Programming: Advances in Theory and Implementation (ASP 2005). Volume 142 of CEUR Workshop Proceedings, Aachen, Germany, CEUR-WS.org (2005)
6. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1) (2009) 1–56
7. Syrjänen, T.: Debugging inconsistent answer-set programs. In: Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR 2006). Volume IfI-06-04 of IfI Technical Report Series, Clausthal-Zellerfeld, Germany, Institut für Informatik, Technische Universität Clausthal (2006) 77–83
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). Volume 4483 of Lecture Notes in Computer Science, Springer (2007) 31–43

9. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Proceedings of the 25th International Conference on Logic Programming (ICLP 2009). Volume 5649 of Lecture Notes in Computer Science, Springer (2009) 296–311
10. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4–5) (2010) 513–529
11. Niemelä, I., Janhunen, T., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010). (2010) 951–956
12. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: Random vs. structure-based testing of answer-set programs: An experimental comparison. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 242–247
13. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. Journal of Artificial Intelligence Research **35** (August 2009) 813–857
14. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* environment. In: Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming (SEA 2007). (2007) 71–85
15. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 317–330
16. iGROM: `http://igrom.sourceforge.net/`
17. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set programs—Project description. In: Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010). Volume 7 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
18. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). (2007) 266–271
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
20. Syrjänen, T.: Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
21. Cliffe, O., De Vos, M., Brain, M., Padget, J.A.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Proceedings of the 24th International Conference on Logic Programming, (ICLP 2008). (2008) 724–728
22. Wittocx, J.: IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation` (2009)
23. Kloimüllner, C., Oetsch, J., Pührer, J., Tompits, H.: `Kara` - A system for visualising and visual editing of interpretations for answer-set programs. In: Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011). (2011)
24. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf (May 2007)
25. Ierusalimschy, R.: Programming in Lua, Second Edition. Lua.Org (2006)
26. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 134–147
27. Boley, H., Kifer, M., eds.: RIF Framework for Logic Dialects. W3C (2010) W3C Recommendation 22 June 2010.
28. M., K., Heymans, S.: RIF core answer set programming dialect. `http://ruleml.org/rif/RIF-CASPD.html` (2009)
29. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proceedings of the 25th Italian Conference on Computational Logic (CILC 2010). (2010)
30. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of the 2nd International Workshop on Software Engineering for Answer-Set Programming (SEA 2009), Potsdam, Germany. (2009)
31. Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: spock: A debugging support tool for logic programs under the answer-set semantics. In: Revised Selected Papers of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and the 21st Workshop on Logic Programming (WLP 2007). Volume 5437 of Lecture Notes in Computer Science, Springer (2009) 247–252
32. ontoprise GmbH: OntoStudio 3.0. (2010) `http://help.ontoprise.de/`.
33. Ricca, F., Gallucci, L., Schindlauer, R., Dell'armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based System for Enterprise Ontologies. Journal of Logic and Computation (2008)

Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs*

Christian Kloimüllner[1], Johannes Oetsch[2], Jörg Pührer[2], and Hans Tompits[2]

[1] Forschungsgruppe für Industrielle Software (INSO),
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
christian.kloimuellner@inso.tuwien.ac.at
[2] Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. In answer-set programming (ASP), the solutions of a problem are encoded in dedicated models, called *answer sets*, of a logical theory. These answer sets are computed from the program that represents the theory by means of an ASP solver and returned to the user as sets of ground first-order literals. As this type of representation is often cumbersome for the user to interpret, tools like ASPVIZ and IDPDraw were developed that allow for visualising answer sets. The tool Kara, introduced in this paper, follows these approaches, using ASP itself as a language for defining visualisations of interpretations. Unlike existing tools that position graphic primitives according to static coordinates only, Kara allows for more high-level specifications, supporting graph structures, grids, and relative positioning of graphical elements. Moreover, generalising the functionality of previous tools, Kara provides modifiable visualisations such that interpretations can be manipulated by graphically editing their visualisations. This is realised by resorting to abductive reasoning techniques. Kara is part of SeaLion, a forthcoming integrated development environment (IDE) for ASP.

1 Introduction

Answer-set programming (ASP) [1] is a well-known paradigm for declarative problem solving. Its key idea is that a problem is encoded in terms of a logic program such that dedicated models of it, called *answer sets*, correspond to the solutions of the problem. Answer sets are interpretations, usually represented by sets of ground first-order literals.

A problem often faced when developing answer-set programs is that interpretations returned by an ASP solver are cumbersome to read—in particular, in case of large interpretations which are spread over several lines on the screen or the output file. Hence, a user may have difficulties extracting the information he or she is interested in from the textual representation of an answer set. Related to this issue, there is one even harder practical problem: editing or writing interpretations by hand.

Although the general goal of ASP is to have answer sets computed automatically, we identify different situations during the development of answer-set programs in which it would be helpful to have adequate means to manipulate interpretations. First, in declarative debugging [2], the user has to specify the semantics he or she expects in order for the debugging system to identify the causes for a mismatch with the actual semantics. In previous work [3], a debugging approach has been introduced that takes a program P and an interpretation I that is expected to be an answer set of P and returns reasons why I is not an answer set of P. Manually producing such an intended interpretation ahead of computation is a time-consuming task, however. Another situation in which the creation of an interpretation can be useful is testing post-processing tools. Typically, if answer-set solvers are used within an online application, they are embedded as a module in a larger context. The overall application delegates a problem to the solver by transforming it to a respective answer-set program and the outcome of the solver is then processed further as needed by the application. In order to test post-processing components, which may be written by programmers unaware

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

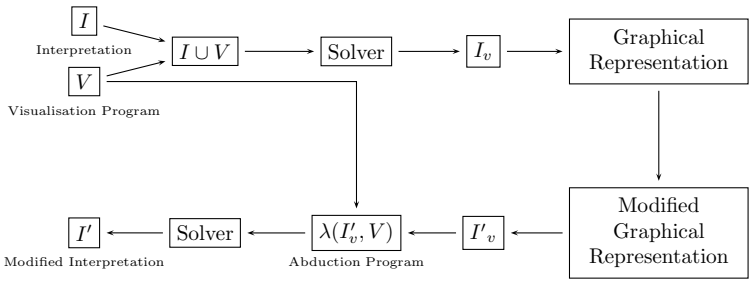


Fig. 1. Overview of the workflow (visualisation and abduction process).

of ASP, it would be beneficial to have means to create mock answer sets as test inputs. Third, the same idea of providing test input applies to modular answer-set programming [4], when a module B that depends on another module A is developed before or separately from A. In order to test B, it can be joined with interpretations mocking answer sets from A.

In this paper, we describe the system Kara which allows for both visualising interpretations and editing them by manipulating their visualisations.[3] The visualisation functionality of Kara has been inspired by the existing tools ASPVIZ [5] and IDPDraw [6] for visualising answer sets. The key idea is to use ASP itself as a language for specifying how to visualise an interpretation I. To this end, the user takes a dedicated answer-set program V—which we call a *visualisation program*—that specifies how the visualisation of I should look like. That is, V defines how different graphical elements, such as rectangles, polygons, images, graphs, etc., should be arranged and configured to visually represent I.

Kara offers a rich visualisation language that allows for defining a superset of the graphical elements available in ASPVIZ and IDPDraw, e.g., providing support for automatically layouting graph structures, relative and absolute positioning, and support for grids of graphical elements. Moreover, Kara also offers a *generic mode* of visualisation, not available in previous tools, that does not require a domain-specific visualisation program, representing an answer set as a hypergraph whose set of nodes corresponds to the individuals occurring in the interpretation.[4] A general difference to previous tools is that Kara does not just produce image files right away but presents the visualisation in form of modifiable graphical elements in a visual editor. The user can manipulate the visualisation in various ways, e.g., change size, position, or other properties of graphical elements, as well as copy, delete, and insert new graphical elements. Notably, the created visualisations can also be used outside our editing framework, as Kara offers an SVG export function that allows to save the possibly modified visualisation as a vector graphic. Besides fine-tuning exported SVG files, manipulation of the visualisation of an interpretation I can be done for obtaining a modified version I' of I by means of abductive reasoning [7]. This gives the possibility to visually edit interpretations which is useful for debugging and testing purposes as described above.

In Section 3, we present a number of examples that illustrate the functionality of Kara and the ease of coping with a visualised answer set compared to interpreting its textual representation.

Kara is designed as a plugin of SeaLion, an Eclipse-based integrated development environment (IDE) for ASP [8] that is currently developed as part of a project on programming-support methods for ASP [9].

2 System Overview

We assume familiarity with the basic concepts of answer-set programming (ASP) (for a thorough introduction to the subject, cf. Baral [1]). In brief, an answer-set program consists of rules of the form

$$a_1 \vee \cdots \vee a_l :- a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n,$$

[3] The name "Kara" derives, with all due respect, from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

[4] A detailed overview of the differences concerning the visualisation capabilities of Kara with other tools is given in Section 4.

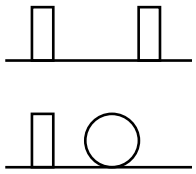


Fig. 2. The visualisation of interpretation I from Example 1.

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, and all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol, $\neg$). For a rule r as above, we define the *head* of r as $\mathrm{H}(r) = \{a_1, \ldots, a_l\}$ and the *positive body* as $\mathrm{B}^+(r) = \{a_{l+1}, \ldots, a_m\}$. If $n = l = 1$, r is a *fact*, and if $l = 0$, r is a *constraint*. For facts, we will usually omit the symbol "$:-$". The *grounding* of a program P relative to its Herbrand universe is defined as usual. An *interpretation* I is a finite and consistent set of ground literals, where consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a. I is an *answer set* of a program P if it is a minimal model of the grounding of the *reduct* of P relative to I (see Baral [1] for details).

The overall workflow of `Kara` is depicted in Fig. 1, illustrating how an interpretation I can be visualised in the upper row and how changing the visualisation can be reflected back to I such that we obtain a modified version I' of I in the lower row. In the following, we call programs that encode problems for which I and I' provide solution candidates *domain programs*.

2.1 Visualisation of Interpretations

As discussed in the introduction, we use ASP itself as a language for specifying how to visualise an interpretation. In doing so, we follow a similar approach as the tools `ASPVIZ` [5] and `IDPDraw` [6]. We next describe this method on an abstract level.

Assume we want to visualise an interpretation I that is defined over a first-order alphabet $\mathcal{A}$. We join I, interpreted as a set of facts, with a visualisation program V that is defined over $\mathcal{A}' \supset \mathcal{A}$, where $\mathcal{A}'$ may contain auxiliary predicates and function symbols, as well as predicates from a fixed set $\mathcal{P}_v$ of reserved *visualisation predicates* that vary for the three tools.[5]

The rules in V are used to derive different atoms with predicates from $\mathcal{P}_v$, depending on I, that control the individual graphical elements of the resulting visualisation including their presence or absence, position, and all other properties. An actual visualisation is obtained by post-processing an answer set I_v of $V \cup I$ that is projected to the predicates in $\mathcal{P}_v$. We refer to I_v as a *visualisation answer set* for I. The process is depicted in the upper row of Fig. 1. An exhaustive list of visualisation predicates available in `Kara` is given in Appendix A.

Example 1. Assume we deal with a domain program whose answer sets correspond to arrangements of items on two shelves. Consider the interpretation $I = \{book(s_1, 1), book(s_1, 3), book(s_2, 1), globe(s_2, 2)\}$ stating that two books are located on shelf s_1 in positions 1 and 3 and that there is another book and a globe on shelf s_2 in positions 1 and 2. The goal is to create a simple graphical representation of this and similar interpretations, depicting the two shelves as two lines, each book as a rectangle, and globes as circles. Consider the following visualisation program:

$$visline(shelf_1, 10, 40, 80, 40, 0). \quad (1)$$
$$visline(shelf_2, 10, 80, 80, 80, 0). \quad (2)$$
$$visrect(f(X, Y), 20, 8) :- book(X, Y). \quad (3)$$
$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- book(s_1, Y). \quad (4)$$
$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- book(s_2, Y). \quad (5)$$
$$visellipse(f(X, Y), 20, 20) :- globe(X, Y). \quad (6)$$
$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- globe(s_1, Y). \quad (7)$$
$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- globe(s2, Y). \quad (8)$$

[5] Technically, in `ASPVIZ`, V is not joined with I but with a domain program P such that I is an answer set of P.

Rules (1) and (2) create two lines with the identifiers $shelf_1$ and $shelf_2$, representing the top and bottom shelf. The second to fifth arguments of $visline/6$ represent the origin and the target coordinates of the line.[6] The last argument of $visline/6$ is a z-coordinate determining which graphical element is visible in case two or more overlap. Rule (3) generates the rectangles representing books, and Rules (4) and (5) determine their position depending on the shelf and the position given in the interpretation. Likewise, Rules (6) to (8) generate and position globes. The resulting visualisation of I is depicted in Fig. 2. □

Note that the first argument of each visualisation predicate is a unique identifier for the respective graphical element. By making use of function symbols with variables, like $f(X, Y)$ in Rule (3) above, these labels are not limited to constants in the visualisation program but can be generated on the fly, depending on the interpretation to visualise. While some visualisation predicates, like $visline$, $visrect$, and $visellipse$, define graphical elements, others, e.g., $visposition$, are used to change properties of the elements, referring to them by their respective identifiers.

Kara also offers a *generic visualisation* that visualises an arbitrary interpretation without the need for defining a visualisation program. In such a case, the interpretation is represented as a labelled hypergraph. Its nodes are the individuals appearing in the interpretation and the edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of the edge are used for expressing the term position of the individual. To distinguish between different predicates, each edge has an additional label stating the predicate. Edges of the same predicate are of the same colour. A generic visualisation is presented in Example 4.

2.2 Editing of Interpretations

We next describe how we can obtain a modified version I' of an interpretation I corresponding to a manipulation of the visualisation of I. We follow the steps depicted in the lower row of Fig. 1, using abductive reasoning. Recall that abduction is the process of finding hypotheses that explain given observations in the context of a theory. Intuitively, in our case, the theory is the visualisation program, the observation is the modified visualisation of I, and the desired hypothesis is I'.

In Kara, the visualisation of I is created using the Graphical Editing Framework (GEF) [10] of Eclipse. It is displayed in a graphical editor which allows for various kinds of manipulation actions such as moving, resizing, adding or deleting graphical elements, adding or removing edges between them, editing their properties, or change grid values. Each change in the visual editor of Kara is internally reflected by a modification to the underlying visualisation answer set I_v. We denote the resulting visualisation interpretation by I'_v. From that and the visualisation program V, we construct a logic program $\lambda(I'_v, V)$ such that the visualisation of any answer set I' of $\lambda(I'_v, V)$ using V corresponds to the modified one.

The idea is that $\lambda(I'_v, V)$, which we refer to as the *abduction program* for I'_v and V, guesses a set of *abducible atoms*. On top of these atoms, the rules of V are used in $\lambda(I'_v, V)$ to derive a hypothetical visualisation answer set I''_v for I'. Finally, constraints in the abduction program ensure that I''_v coincides with the targeted visualisation interpretation I'_v on a set $\mathcal{P}_i$ of selected predicates from $\mathcal{P}_v$, which we call *integrity predicates*. Hence, a modified interpretation I' can be obtained by computing an answer set of $\lambda(I'_v, V)$ and projecting it to the guessed atoms. To summarise, the abduction problem underlying the described process can be stated as follows:

(∗) Given the interpretation I'_v, determine an interpretation I' such that I'_v coincides with each answer set of $V \cup I'$ on $\mathcal{P}_i$.

Clearly, visualisation programs must be written in a way that manipulated visualisation interpretations could indeed be the outcome of the visualisation program for some input. This is not the case for arbitrary visualisation programs, but usually it is easy to write an appropriate visualisation program that allows for abducing interpretations.

The following problems have to be addressed for realising the sketched approach:

– determining the predicates and domains of the abducible atoms, and

[6] The origin of the coordinate system is at the top-left corner of the illustration window with the x-axis pointing to the right and the y-axis pointing down.

$$
\begin{aligned}
\mathsf{dom}(I'_v, V) = \{&\mathit{nonRecDom}(t) :- v(\boldsymbol{t'}) \mid r \in V, v/m \in \mathcal{P}_v, v(\boldsymbol{t'}) \in \mathrm{H}(r),\\
&\quad a(\boldsymbol{t}) \in \mathrm{B}^+(r), \boldsymbol{t} = t_1, \ldots, t, \ldots, t_n, a/n \notin \mathcal{P}_v,\\
&\quad \mathit{VAR}(t) \neq \emptyset, \mathit{VAR}(t) \subseteq \mathit{VAR}(\boldsymbol{t'})\}\cup\\
&\{\mathit{dom}(t) :- v(\boldsymbol{t'}), \mathit{nonRecDom}(X_1), \ldots, \mathit{nonRecDom}(X_l) \mid r \in V,\\
&\quad v/m \in \mathcal{P}_v, v(\boldsymbol{t'}) \in \mathrm{H}(r), a(\boldsymbol{t}) \in \mathrm{B}^+(r), \boldsymbol{t} = t_1, \ldots, t, \ldots, t_n,\\
&\quad a/n \notin \mathcal{P}_v, \mathit{VAR}(t) \cap \mathit{VAR}(\boldsymbol{t'}) \neq \emptyset,\\
&\quad \mathit{VAR}(t) \setminus \mathit{VAR}(\boldsymbol{t'}) = \{X_1, \ldots, X_l\}\}\cup\\
&\{\mathit{dom}(X) :- \mathit{nonRecDom}(X)\},
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{guess}(V) = \{&a(X_1, \ldots, X_n) :- \text{not } \neg a(X_1, \ldots, X_n), \mathit{dom}(X_1), \ldots, \mathit{dom}(X_n),\\
&\neg a(X_1, \ldots, X_n) :- \text{not } a(X_1, \ldots, X_n), \mathit{dom}(X_1), \ldots, \mathit{dom}(X_n) \mid\\
&\quad a/n \notin \mathcal{P}_v, a(t_1, \ldots, t_n) \in \textstyle\bigcup_{r \in V} \mathrm{B}(r),\\
&\quad \{a(t'_1, \ldots, t'_n) \mid a(t'_1, \ldots, t'_n) \in \mathrm{H}(r), r \in V\} = \emptyset\},
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{check}(I'_v) = \{&:- \text{not } v(t_1, \ldots, t_n),\ :- v(X_1, \ldots, X_n), \text{not } v'(X_1, \ldots, X_n),\\
&v'(t_1, \ldots, t_n) \mid v(t_1, \ldots, t_n) \in I'_v, v/n \in \mathcal{P}_i\},
\end{aligned}
$$

Fig. 3. Elements of the abduction program $\lambda(I'_v, V)$.

– choosing the integrity predicates among the visualisation predicates.

For solving these issues, we rely on pragmatic choices that seem useful in practice. We obtain the set $\mathcal{P}_a$ of predicates of the abducible atoms from the visualisation program V. The idea is that every predicate that is relevant to the solution of a problem encoded in an answer set has to occur in the visualisation program if the latter is meant to provide a complete graphical representation of the solution. Moreover, we restrict $\mathcal{P}_a$ to those non-visualisation predicates in V that occur in the body of a rule but not in any head atom in V. The assumption is that atoms defined in V are most likely of auxiliary nature and not contained in a domain program.

An easy approach for generating a domain $\mathcal{D}_a$ of the abducible atoms would be to extract the terms occurring in I'_v. We follow, however, a more fine-grained approach that takes the introduction and deletion of function symbols in the rules in V into account. Assume V contains the rules

$$
\begin{aligned}
\mathit{visrect}(f(\mathit{Street}, \mathit{Num}), 9, 10) :-&\ \mathit{house}(\mathit{Street}, \mathit{Num}) \quad \text{and}\\
\mathit{visellipse}(\mathit{sun}, \mathit{Width}, \mathit{Height}) :-&\ \mathit{property}(\mathit{sun}, \mathit{size}(\mathit{Width}, \mathit{Height})),
\end{aligned}
$$

and I'_v contains $\mathit{visrect}(f(\mathit{bakerstreet}, 221b), 9, 10)$ and $\mathit{visellipse}(\mathit{sun}, 10, 11)$. Then, when extracting the terms in I'_v, the domain includes $f(\mathit{bakerstreet}, 221b)$, $\mathit{bakerstreet}$, $221b$, 9, 10, sun, and 11 for the two rules. However, the functor f is solely an auxiliary concept in V and not meant to be part of domain programs. Moreover, the term 9 is introduced in V and is not needed in the domain for I'. Also, the terms 10 and 11 as standalone terms and sun are not needed in I' to derive I'_v. Even worse, the term $\mathit{size}(10, 11)$, that has to be contained in I' such that I'_v can be a visualisation answer set for I', is missing in the domain. Hence, we derive $\mathcal{D}_a$ in $\lambda(I'_v, V)$ not only from I'_v but also consider the rules in V. Using our translation that is detailed below, we obtain $\mathit{bakerstreet}$, $221b$, and $\mathit{size}(10, 12)$ as domain terms from the rules above.

For the choice of $\mathcal{P}_i$, i.e., of the predicates on which I'_v and the actual visualisation answer sets of I' need to coincide, we exclude visualisation predicates that require a high preciseness in visual editing by the user in order to match exactly a value that could result from the visualisation program. For example, we do not include predicates determining position and size of graphical elements, since in general it is hard to position and scale an element precisely such that an interpretation I' exists with a matching visualisation. Note that this is not a major restriction, as in general it is easy to write a visualisation program such that aspects that the user wants to be modifiable are represented by graphical elements that can be elegantly modified visually. For example, instead of representing a Sudoku puzzle by labels whose exact position is calculated in the visualisation program, the language of `Kara` allows for using a logical grid such that the value of each cell can be easily changed in the visual editor.

We next give the details of the abduction program.

Definition 1. *Let I'_v be an interpretation with atoms over predicates in $\mathcal{P}_v$, V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Moreover, let $VAR(T)$ denote the variables occurring in T, where T is a term or a list of terms. Then, the* abduction program *with respect to I'_v and V is given by*

$$\lambda(I'_v, V) = \mathsf{dom}(I'_v, V) \cup \mathsf{guess}(V) \cup V \cup \mathsf{check}(I'_v),$$

where $\mathsf{dom}(I'_v, V)$, $\mathsf{guess}(V)$, and $\mathsf{check}(I'_v)$ are given in Fig. 3, and $nonRecDom/1$, $dom/1$, and v'/n, for all $v/n \in \mathcal{P}_i$, are fresh predicates.

The idea of $\mathsf{dom}(I'_v, V)$ is to consider non-ground terms t contained in the body of a visualisation rule that share variables with a visualisation atom in the head of the rule and to derive instances of these terms when the corresponding visualisation atom is contained in I'_v. In case less variables occur in the visualisation atom than in t, we avoid safety problems by restricting their scope to parts of the derived domain. Here, the distinction between predicates dom and $nonRecDom$ is necessary to prevent infinite groundings of the abduction program. Note that in general it is not guaranteed that the domain we derive contains all necessary elements for abducing an appropriate interpretation I'. For instance, consider the case that the visualisation program contains a rule $visrect(id, 5, 5) :- foo(X)$, and V together with the constraints in $\mathsf{check}(I'_v)$ require that for all terms t of a domain that can be obtained from I'_v and V, $foo(t)$ must not hold. Then, there is no interpretation that will trigger the rule using this domain, although an interpretation with a further term t' might exist that results in the desired visualisation. Hence, we added an editor to Kara that allows for changing and extending the automatically generated domain as well as the set of abducible predicates.

The following result characterises the answer sets of the abduction program.

Theorem 1. *Let I'_v be an interpretation with atoms over predicates in $\mathcal{P}_v$, V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Then, any answer set I''_v of $\lambda(I'_v, V)$ coincides with I'_v on the atoms over predicates from $\mathcal{P}_i$, and a solution I' of the abduction problem $(*)$ is obtained from I''_v by projection to the predicates in*

$$\{a/n \mid a(t_1, \ldots, t_n) \in \bigcup_{r \in V} \mathrm{B}(r), \{a(t'_1, \ldots, t'_n) \mid a(t'_1, \ldots, t'_n) \in \mathrm{H}(r), r \in V\} = \emptyset\} \setminus \mathcal{P}_v.$$

2.3 Integration in SeaLion

Kara is written in Java and integrated in the Eclipse-plugin SeaLion [8] for developing answer-set programs. Currently, it can be used with answer-set programs in the languages of Gringo and DLV. SeaLion offers functionality to execute external ASP solvers on answer-set programs. The resulting answer sets can be parsed by the IDE and displayed as expandable tree structures in a dedicated Eclipse view for interpretations. Starting from there, the user can invoke Kara by choosing a pop-up menu entry of the interpretation he or she wants to visualise. A run configuration dialog will open that allows for choosing the visualisation program and for setting the solver configuring to be used by Kara. Then, the visual editor opens with the generated visualisation. The process for abducing an interpretation that reflects the modifications to the visualisation can be started from the visual editor's pop-up menu. If a respective interpretation exists, one will be added to SeaLion's interpretation view.

The sources of Kara and the alpha version of SeaLion can be downloaded from

`http://sourceforge.net/projects/mmdasp/`.

An Eclipse update site will be made available as soon as SeaLion reaches beta status.

3 Examples

In this section, we provide examples that give an overview of Kara's functionality. We first illustrate the use of logic grids and the visual editing feature.

```
visgrid(maze, MAXR, MAXC, MAXR*20+5, MAXC*20+5) :- maxC(MAXC), maxR(MAXR). (9)
visposition(maze, 0, 0, 0). (10)
% A cell with a wall on it.
visrect(wall, 20, 20). (11)
visbackgroundcolor(wall, black). (12)
% An empty cell.
visrect(empty, 20, 20). (13)
visbackgroundcolor(empty, white). (14)
viscolor(empty, white). (15)
% Entrance and exit.
visimage(entrance, "entrance.jpg"). (16)
visscale(entrance, 18, 18). (17)
visimage(exit, "exit.png"). (18)
visscale(exit, 18, 18). (19)
% Filling the cells of the grid.
visfillgrid(maze, empty, R, C) :- empty(C, R), not entrance(C, R), not exit(C, R). (20)
visfillgrid(maze, wall, R, C) :- wall(C, R), not entrance(C, R), not exit(C, R). (21)
visfillgrid(maze, entrance, R, C) :- entrance(C, R). (22)
visfillgrid(maze, exit, R, C) :- exit(C, R). (23)
% Vertical and horizontal lines.
visline(v(0), 5, 5, 5, MAXR * 20 + 5, 1) :- maxR(MAXR). (24)
visline(v(C), C*20+5, 5, C*20+5, MAXR*20+5, 1) :- col(C), maxR(MAXR). (25)
visline(h(0), 5, 5, MAXC * 20 + 5, 5, 1) :- maxC(MAXC). (26)
visline(h(R), 5, R * 20 + 5, MAXC * 20 + 5, R * 20 + 5, 1) :- row(R), maxC(MAXC). (27)
% Define possible grid values for editing.
vispossiblegridvalues(maze, wall). (28)
vispossiblegridvalues(maze, empty). (29)
vispossiblegridvalues(maze, entrance). (30)
vispossiblegridvalues(maze, exit). (31)
```

Fig. 4. Visualisation program for Example 2.

Example 2. Maze-generation is a benchmark problem from the second ASP competition [11]. The task is to generate a two-dimensional grid, where each cell is either a wall or empty, that satisfies certain constraints. There are two dedicated empty cells, being the maze's entrance and its exit, respectively. The following facts represent a sample answer set of a maze generation encoding restricted to interesting predicates.

$$col(1..5).\ row(1..5).\ maxC(5).\ maxR(5).\ wall(1,1).\ empty(1,2).\ wall(1,3).\ wall(1,4).\ wall(1,5).\ wall(2,1).\ empty(2,2).\ empty(2,3).\ empty(2,4).\ wall(2,5).\ wall(3,1).\ wall(3,2).\ wall(3,3).\ empty(3,4).\ wall(3,5).\ wall(4,1).\ empty(4,2).\ empty(4,3).\ empty(4,4).\ wall(4,5).\ wall(5,1).wall(5,2).\ wall(5,3).\ empty(5,4).\ wall(5,5).\ entrance(1,2).\ exit(5,4).$$

Predicates $col/1$ and $row/1$ define indices for the rows and columns of the maze, while $maxC/1$ and $maxR/1$ give the maximum column and row number, respectively. The predicates $wall/2$, $empty/2$, $entrance/2$, and $exit/2$ determine the positions of walls, empty cells, the entrance, and the exit in the grid, respectively. One may use the visualisation program from Fig. 4 for maze-generation interpretations of this kind.

In Fig. 4, Rule (9) defines a logic grid with identifier $maze$, $MAXR$ rows, and $MAXC$ columns. The fourth and fifth parameter define the height and width of the grid in pixel. Rule (10) is a fact that defines a fixed position for the maze. The next step is to define the graphical objects to be displayed in the grid. Because these objects are fixed (i.e., they are used more than once), they can be defined as facts. A wall is

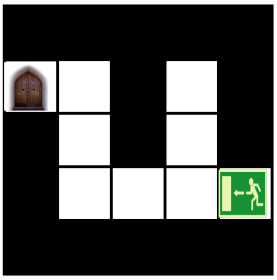


Fig. 5. Visualisation output for the maze-generation program.

represented by a rectangle with black background and foreground colour[7] (Rules (11) and (12)) whereas an empty cell is rendered as a rectangle with white background and foreground colour (Rules (13) to (15)). The entrance and the exit are represented by two images (Rules (16) to (19)). Then, these graphical elements are assigned to the respective cell of the grid (Rules (20) to (23)). Rules (24) to (27) render vertical and horizontal lines to better distinguish between the different cells. Rules (28) to (31) are not needed for visualisation but define possible values for the grid that we want to be available in the visual editor.

Once the grid is rendered, the user can replace the value of a cell with a value defined using predicate $vispossiblegridvalues/2$ (e.g., replacing an empty cell with a wall). The visualisation of the sample interpretation using this program is given in Fig. 5. Note that the visual representation of the answer set is much easier to cope with than the textual representation of the answer set given in the beginning of the example.

Next, we demonstrate how to use the visual editing feature of Kara to obtain a modified interpretation, as shown in Fig. 6. Suppose we want to change the cell $(3, 2)$ from being a wall to an empty cell. The user can select the respective cell and open a pop-up menu that provides an item for changing grid-values. A dialog opens that allows for choosing among the values that have been defined in the visualisation program, using the $vispossiblegridvalues/2$ predicate. When the user has finished editing the visualisation, he or she can start the abduction process for inferring the new interpretation. When an interpretation is successfully derived, it is added to SeaLion's interpretation view. □

Kara supports absolute and relative positioning of graphical elements. If for any visualisation element the predicate $visposition/4$ is defined, then we have fixed positioning. Otherwise, the element is positioned automatically. Then, by default, the elements are randomly positioned on the graphical editor. However, the user can define the position of an element *relative* to another element. This is done by using the predicates $visleft/2$, $visright/2$, $visabove/2$, $visbelow/2$, and $visinfrontof/2$.

Example 3. The following visualisation program makes use of relative positioning for sorting elements according to their label.

$$visrect(a, 50, 50). \tag{32}$$
$$vislabel(a, laba). \tag{33}$$
$$vistext(laba, 3). \tag{34}$$
$$vispolygon(b, 0, 20, 1). \tag{35}$$
$$vispolygon(b, 25, 0, 2). \tag{36}$$
$$vispolygon(b, 50, 20, 3). \tag{37}$$
$$vislabel(b, labb). \tag{38}$$
$$vistext(labb, 10). \tag{39}$$
$$visellipse(c, 30, 30). \tag{40}$$
$$vislabel(c, labc). \tag{41}$$
$$vistext(labc, 5). \tag{42}$$
$$element(X) :- visrect(X, \_, \_). \tag{43}$$
$$element(X) :- vispolygon(X, \_, \_, \_). \tag{44}$$
$$element(X) :- visellipse(X, \_, \_).element \tag{45}$$

[7] Black foreground colour is default and may not be set explicitly.

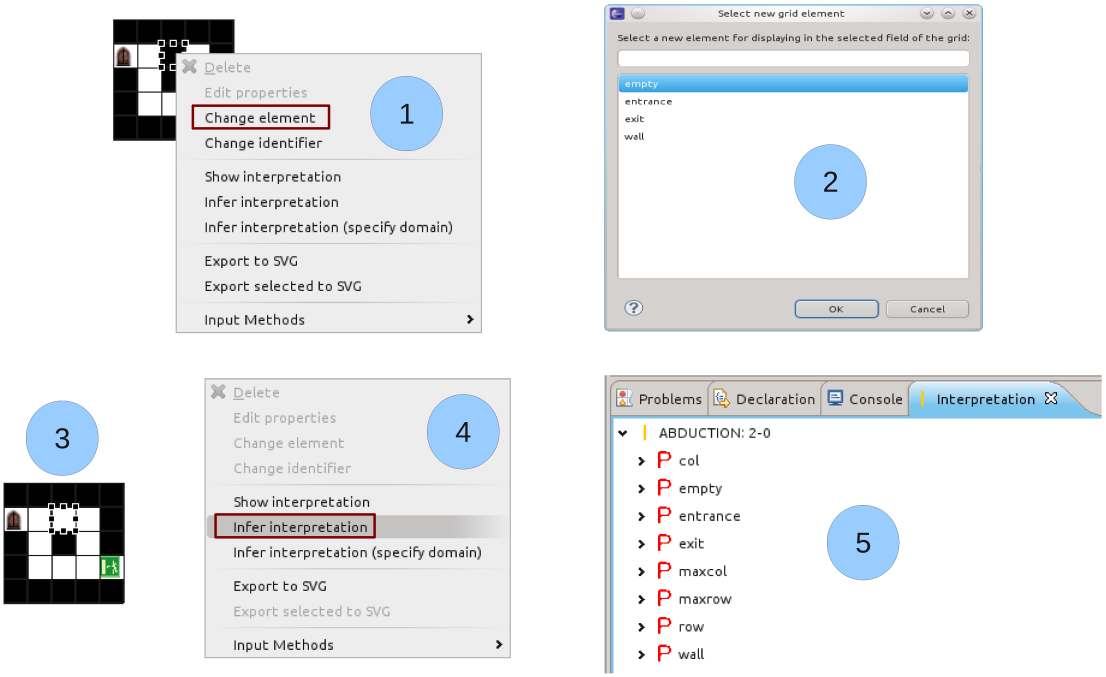


Fig. 6. Abduction steps in the plugin.

$$\begin{aligned} \mathit{visleft}(X,Y) :- \ & \mathit{element}(X), \mathit{element}(Y), \mathit{vislabel}(X, \mathit{LABX}), \\ & \mathit{vistext}(\mathit{LABX}, \mathit{XNUM}), \mathit{vislabel}(Y, \mathit{LABY}), \\ & \mathit{vistext}(\mathit{LABY}, \mathit{YNUM}), \mathit{XNUM} < \mathit{YNUM}. \end{aligned} \tag{46}$$

The program defines three graphical objects, a rectangle, a polygon, and an ellipse. In Rules (32) to (34), the rectangle together with its label 3 is generated. The shape of the polygon (Rules (35) to (37)) is defined by a sequence of points relative to the polygon's own coordinate system using the $\mathit{vispolygon}/4$ predicate. The order in which these points are connected with each other is given by the predicate's fourth argument. Rules (38) and (39) generate the label for the polygon and specify its text. Rules (43) to (45) state that every rectangle, polygon, and ellipse is an element. The relative position of the three elements is determined by Rule (46). For two elements E_1 and E_2, E_1 has to appear to the left of E_2 whenever the label of E_1 is smaller than the one of E_1. The output of this visualisation program is given in Fig. 7. Note that the visualisation program does not make reference to predicates from an interpretation to visualise, hence the example illustrates that `Kara` can also be used for creating arbitrary graphics. □

The last example demonstrates the support for graphs in `Kara`. Moreover, the generic visualisation feature is illustrated.

Example 4. We want to visualise answer sets of an encoding of a graph-colouring problem. Assume we have the following interpretation that defines nodes and edges of a graph as well as a colour for each node.

$$\begin{aligned} \{ & \mathit{node}(1),\ \mathit{node}(2),\ \mathit{node}(3),\ \mathit{node}(4),\ \mathit{node}(5),\ \mathit{node}(6),\ \mathit{edge}(1,2),\ \mathit{edge}(1,3), \\ & \mathit{edge}(1,4),\ \mathit{edge}(2,4),\ \mathit{edge}(2,5),\ \mathit{edge}(2,6),\ \mathit{edge}(3,1),\ \mathit{edge}(3,4),\ \mathit{edge}(3,5), \\ & \mathit{edge}(4,1),\ \mathit{edge}(4,2),\ \mathit{edge}(5,3),\ \mathit{edge}(5,4),\ \mathit{edge}(5,6),\ \mathit{edge}(6,2),\ \mathit{edge}(6,3), \\ & \mathit{edge}(6,5),\ \mathit{color}(1,\mathit{lightblue}),\ \mathit{color}(2,\mathit{yellow}),\ \mathit{color}(3,\mathit{yellow}),\ \mathit{color}(4,\mathit{red}), \\ & \mathit{color}(5,\mathit{lightblue}),\ \mathit{color}(6,\mathit{red})\}. \end{aligned}$$

We make use of the following visualisation program:

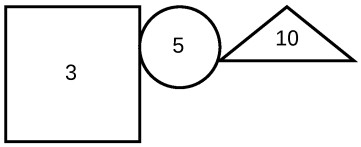


Fig. 7. Output of the visualisation program in Example 3.

```
% Generate a graph.
```
$visgraph(g).$ (47)
```
% Generate the nodes of the graph.
```
$visellipse(X, 20, 20) :- node(X).$ (48)
$visisnode(X, g) :- node(X).$ (49)
```
% Connect the nodes (edges of the input).
```
$visconnect(f(X, Y), X, Y) :- edge(X, Y).$ (50)
$vistargetdeco(X, arrow) :- visconnect(X, \_, \_).$ (51)
```
% Generate labels for the nodes.
```
$vislabel(X, l(X)) :- node(X).$ (52)
$vistext(l(X), X) :- node(X).$ (53)
$visfontstyle(l(X), bold) :- node(X).$ (54)
```
% Color the node according to the solution.
```
$visbackgroundcolor(X, COLOR) :- node(X), color(X, COLOR).$ (55)

In Rule (47), a graph, g, is defined and a circle for every node from the input interpretation is created (Rule (48)). Rule (49) states that each of these circles is logically considered a node of graph g. This has the effect that they will be considered by the algorithm layouting the graph during the creation of the visualisation. The edges of the graph are defined using the $visconnect/3$ predicate (Rule (50)). It can be used to connect arbitrary graphical elements with a line, also if they are not nodes of some graph. As we deal with a directed graph, an arrow is set as target decoration for all the connections (Rule (51)). Labels for the nodes are set in Rules (52) to (54). Finally, Rule (55) sets the colour of the node according to the interpretation. The resulting visualisation is depicted in Fig. 8. Moreover, the generic visualisation of the graph colouring interpretation is given in Fig. 9. □

4 Related Work

The visualisation feature of Kara follows the previous systems ASPVIZ [5] and IDPDraw [6], which also use ASP for defining how interpretations should be visualised.[8] Besides the features beyond visualisation, viz. the framework for editing visualisations and the support for multiple solvers, there are also differences between Kara and these tools regarding visualisation aspects.

Kara allows to write more high-level specifications for positioning the graphical elements of a visualisation. While IDPDraw and ASPVIZ require the use of absolute coordinates, Kara additionally supports relative positioning and automatic layouting for graph and grid structures. Note that technically, the former is realised using ASP, by guessing positions of the individual elements and adding respective constraints to ensure the correct layout, while the latter is realised by using a standard graph layouting algorithm which is part of the Eclipse framework. In Kara, as well as in IDPDraw, each graphical element has a unique identifier that can be used, e.g., to link elements or to set their properties (e.g., colour or font style). That way, programs can be written in a clear and elegant way since not all properties of an element have to be specified within a single atom. Here, Kara exploits that the latest ASP solvers support function symbols

[8] IDPDraw has been used for visualisation of the benchmark problems of the second and third ASP competition.

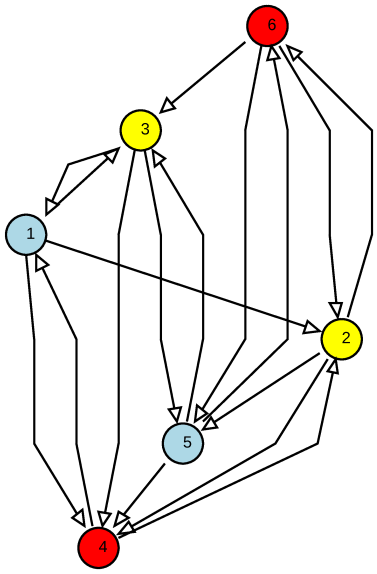


Fig. 8. Visualisation of a coloured graph.

that allow for generating new identifiers from terms of the interpretation to visualise. IDPDraw does not support function symbols. Instead, for having compound identifiers, IDPDraw uses predicates of variable length (e.g., $idp\_polygon(id_1, id_2, ...)$). A disadvantage of this approach is that some solvers, like DLV, do not support predicates of variable length. ASPVIZ does not support identifiers for graphical objects.

The support for a z-axis to determine which object should be drawn over others is available in Kara and IDPDraw but missing in ASPVIZ. Both Kara and ASPVIZ support the export of visualisations as vector graphics in the SVG format, which is not possible with IDPDraw. A feature that is supported by ASPVIZ and IDPDraw, however, is creating animations which is not possible with Kara so far.

Kara and ASPVIZ are written in Java and depend only on a Java Virtual Machine. IDPDraw, on the other hand, is written in C++ and depends on the qt libraries. Finally, Kara is embedded in an IDE, whereas ASPVIZ and IDPDraw are stand-alone tools.

A related approach from software engineering is the Alloy Analyzer, a tool to support the analysis of declarative software models [12]. Models are formulated in a first-order based specification language. The Alloy Analyzer can find satisfying instances of a model using translations to SAT. Instances of models are first-order structures that can be automatically visualised as graphs, where the nodes correspond to atoms from respective signature declarations in the specification, and the edges correspond to relations between atoms. Since the Alloy approach is based on finding models for declarative specifications, it can be regarded as an instance of ASP in a broader sense. The visualisation of first-order structures in Alloy is closely related to the generic visualisation mode of Kara where no dedicated visualisation program is needed. Alloy supports filtering predicates and arguments away of the graph. We consider to add such a feature in future versions of Kara for getting a clearer generic visualisation.

5 Conclusion

We presented the tool Kara for visualising and visual editing of interpretations in ASP. It supports generic as well as customised visualisations. For the latter, a powerful language for defining a visualisation by means of ASP is provided, supporting, e.g., automated graph layouting, grids of graphical elements, and relative positioning. The editing feature is based on abductive reasoning, inferring a new interpretation as hypothesis to explain a modified visualisation. In future work, we want to add support for defining input and output signatures for programs in SeaLion. Then, the abduction framework of Kara could be easily extended such that instead of deriving an interpretation that corresponds to the modified visualisation, one can derive inputs for a domain program such that one of its answer sets has this visualisation.

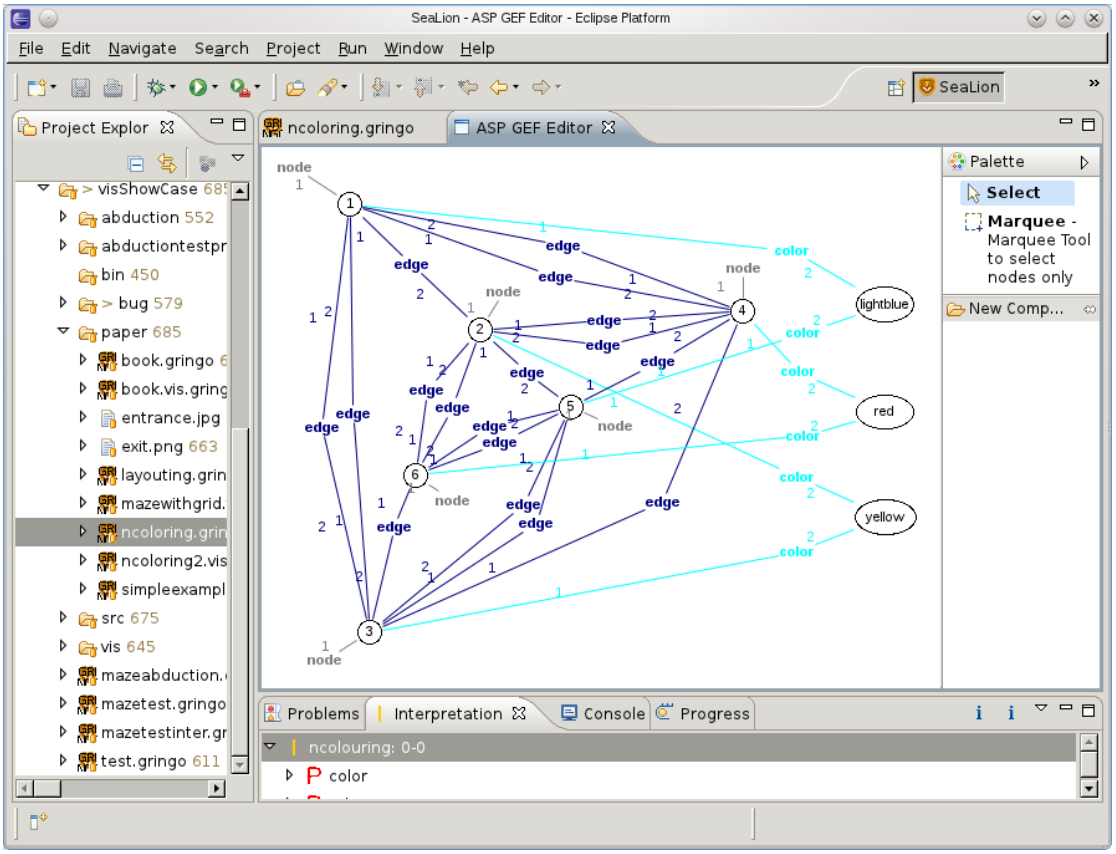


Fig. 9. A screenshot of SeaLion's visual interpretation editor showing a generic visualisation of the graph colouring interpretation of Example 4 (the layout of the graph has been manually optimised by moving the nodes in the editor).

A Predefined Visualisation Predicates in Kara

Atom	Intended meaning
$visellipse(id, height, width)$	Defines an ellipse with specified height and width.
$visrect(id, height, width)$	Defines a rectangle with specified height and width.
$vispolygon(id, x, y, ord)$	Defines a point of a polygon. The ordering defines in which order the defined points are connected with each other.
$visimage(id, path)$	Defines an image given in the specified file.
$visline(id, x_1, y_1, x_2, y_2, z)$	Defines a line between the points (x_1, y_1) and (x_2, y_2).
$visgrid(id, rows, cols, height, width)$	Defines a grid, with the specified number of rows and columns; $height$ and $width$ determine the size of the grid.
$visgraph(id)$	Defines a graph.
$vistext(id, text)$	Defines a text element.
$vislabel(id_g, id_t)$	Sets the text element id_t as a label for graphical element id_g. Labels are supported for the following elements: $visellipse/3$, $visrect/3$, $vispolygon/4$, and $visconnect/3$.
$visisnode(id_n, id_g)$	Adds the graphical element id_n as a node to a graph id_g for automatic layouting. The following elements are supported as nodes: $visrect/3$, $visellipse/3$, $vispolygon/4$, $visimage/2$.
$visscale(id, height, weight)$	Scales an image to the specified height and width.
$visposition(id, x, y, z)$	Puts an element id on the fixed position (x, y, z).

$visfontfamily(id,ff)$	Sets the specified font ff for a text element id.
$visfontsize(id,size)$	Sets the font size $size$ for a text element id.
$visfontstyle(id,style)$	Sets the font style for a text element id to bold or italics.
$viscolor(id,color)$	Sets the foreground colour for the element id.
$visbackgroundcolor(id,color)$	Sets the background colour for the element id.
$visfillgrid(id_g,id_c,row,col)$	Puts element id_c in cell (row, col) of the grid id_g.
$visconnect(id_c,id_{g_1},id_{g_2})$	Connects two elements, id_{g_1} and id_{g_2}, by a line such that id_{g_1} is the source and id_{g_2} is the target of the connection.
$vissourcedeco(id,deco)$	Sets the source decoration for a connection.
$vistargetdeco(id,deco)$	Sets the target decoration for a connection.
$visleft(id_l,id_r)$	Ensures that the x-coordinate of id_l is less than that of id_r.
$visright(id_r,id_l)$	Ensures that the x-coordinate of id_r is greater than that of id_l.
$visabove(id_t,id_b)$	Ensures that the y-coordinate of id_t is smaller than that of id_b.
$visbelow(id_b,id_t)$	Ensures that the y-coordinate of id_b is greater than that of id_t.
$visinfrontof(id_1,id_2)$	Ensures that the z-coordinate of id_1 is greater than that of id_2.
$vishide(id)$	Hides the element id.
$visdeletable(id)$	Defines that the element id can be deleted in the visual editor.
$viscreatable(id)$	Defines that the element id can be created in the visual editor.
$vischangable(id,prop)$	Defines that the property $prop$ can be changed for the element id in the visual editor.
$vispossiblegridvalues(id,id_e)$	Defines that graphical element id_e is available as possible grid value for a grid id in the visual editor.

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England, UK (2003)
2. Shapiro, E.Y.: Algorithmic Program Debugging. PhD thesis, Yale University, New Haven, CT, USA (May 1982)
3. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4–5) (2010) 513–529
4. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. Journal of Artificial Intelligence Research **35** (2009) 813–857
5. Cliffe, O., De Vos, M., Brain, M., Padget, J.A.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Proceedings of the 24th International Conference on Logic Programming, (ICLP 2008). (2008) 724–728
6. Wittocx, J.: IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation` (2009)
7. Peirce, C.S.: Abduction and induction. In: Philosophical Writings of C.S. Peirce, Chapter 11. (1955) 150–156
8. Oetsch, J., Pührer, J., Tompits, H.: The `SeaLion` has landed: An IDE for answer-set programming—Preliminary report. In: Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011). (2011)
9. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set programs—Project description. In Hermenegildo, M., Schaub, T., eds.: Technical Communications of the 26th International Conference on Logic Programming (ICLP'10). Volume 7 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
10. The Eclipse Foundation: Eclipse Graphical Editing Framework. `http://www.eclipse.org/gef/`
11. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In Erdem, E., Lin, F., Schaub, T., eds.: Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Volume 5753 of Lecture Notes in Computer Science, Springer (2009) 637–654
12. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)

Unit Testing in *ASPIDE*

Onofrio Febbraro[1], Nicola Leone[2], Kristian Reale[2], and Francesco Ricca[2]

[1] DLVSystem s.r.l. - P.zza Vermicelli, Polo Tecnologico, 87036 Rende, Italy
febbraro@dlvsystem.com
[2]Dipartimento di Matematica, Università della Calabria, 87036 Rende, Italy
{leone,reale,ricca}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a declarative logic programming formalism, which is employed nowadays in both academic and industrial real-world applications. Although some tools for supporting the development of ASP programs have been proposed in the last few years, the crucial task of *testing* ASP programs received less attention, and is an Achilles' heel of the available programming environments.
In this paper we present a language for specifying and running *unit tests* on ASP programs. The testing language has been implemented in *ASPIDE*, a comprehensive IDE for ASP, which supports the entire life-cycle of ASP development with a collection of user-friendly graphical tools for program composition, *testing*, debugging, profiling, solver execution configuration, and output-handling.

1 Introduction

Answer Set Programming (ASP) [1] is a declarative logic programming formalism proposed in the area of non-monotonic reasoning. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find those solutions [2].

The language of ASP [1] supports a number of modeling constructs including disjunction in rule heads, nonmonotonic negation [1], (weak and strong) constraints [3], aggregate functions [4], and more. These features make ASP very expressive [5], and suitable for developing advanced real-world applications. ASP is employed in several fields, from Artificial Intelligence [6–11] to Information Integration [12], and Knowledge Management [13, 14]. Interestingly, these applications of ASP recently have stimulated some interest also in industry [14].

On the one hand, the effective application of ASP in real-world scenarios was made possible by the availability of efficient ASP systems [6, 18, 19]. On the other hand, the adoption of ASP can be further boosted by offering effective programming tools capable of supporting the programmers in managing large and complex projects [20].

In the last few years, a number of tools for developing ASP programs have been proposed, including editors and debuggers [21–31]. Among them, *ASPIDE* [31] –which stands for Answer Set Programming Integrated Development Environment– is one of the most complete development tools[1] and it integrates a cutting-edge editing tool (featuring dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly graphical tools for program composition, debugging, profiling, DBMS access, solver execution configuration and output-handling.

Although so many tools for developing ASP programs have been proposed up to now, the crucial task of *testing* ASP programs received less attention [32, 46], and is an Achilles' heel of the available programming environments. Indeed, the majority of available graphic programming environments for ASP does not provide the user with a testing tool (see [31]), and also the one present in the first versions of *ASPIDE* is far from being effective.

In this paper we present a pragmatic solution for testing ASP programs. In particular, we present a new language for specifying and running *unit tests* on ASP programs. The testing language presented in this paper is inspired by the JUnit [33] framework: the developer can specify the rules composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the

[1] For an exaustive feature-wise comparison with existing environments for developing logic programs we refer the reader to [31].

expected behavior of sub-programs. The obtained test case specification can be run by exploiting an ASP solver, and the assertions are automatically verified by analyzing the output of the chosen ASP solver. Note that test case specification is applicable independently of the used ASP solver. The testing language was implemented in *ASPIDE*, which also provides the user with some graphic tools that make the development of test cases simpler. The testing tool described in this work extends significantly the one formerly available in *ASPIDE*, by both extending the language by more expressive (non-ground) assertions and the support of weak-constraints, and enriching its collection of user-friendly graphical tools (including program composition, debugging, profiling, database management, solver execution configuration, and output-handling) with a graphical test suite management interface.

As far as related work is concerned, the task of testing ASP programs was approached for the first time, to the best of our knowledge, in [32, 46] where the notion of structural testing for ground normal ASP programs is defined and methods for automatically generating tests is introduced. The results presented in [32, 46] are, somehow, orthogonal to the contribution of this paper. Indeed, no language/implementation is proposed in [32, 46] for specifying/automatically-running the produced test cases; whereas, the language presented in this paper can be used for encoding the output of a test case generator based on the methods proposed in [32]. Finally, it is worth noting that, testing approaches developed for other logic languages, like prolog [34–36], cannot be straightforwardly ported to ASP because of the differences between the languages.

The rest of this paper is organized as follows: in Section 2 we overview *ASPIDE*; in section 3 we introduce a language for specifying unit tests for ASP programs; in Section 4 we describe the user interface components of *ASPIDE* conceived for creating and running tests; finally, in Section 5 we draw the conclusion.

2 *ASPIDE*: Integrated Development Environment for ASP

ASPIDE is an Integrated Development Environment (IDE) for ASP, which features a rich *editing tool* with a collection of user-friendly *graphical tools* for ASP program development. In this section we first summarize the main features of the system and then we overview the main components of the *ASPIDE* user interface. For a more detailed description of *ASPIDE*, as well as for a complete comparison with competing tools, we refer the reader to [31] and to the online manual published in the system web site `http://www.mat.unical.it/ricca/aspide`.

System Features. *ASPIDE* is inspired by Eclipse, one of the most diffused programming environments. The main features of *ASPIDE* are the following:

- *Workspace management*. The system allows one to organize ASP programs in projects, which are collected in a special directory (called workspace).
- *Advanced text editor*. The editing of ASP files is simplified by an advanced text editor. Currently, the system is able to load and store ASP programs in the syntax of the ASP system DLV [15], and supports the `ASPCore` language profile employed in the ASP System Competition 2011 [37]. *ASPIDE* can also manage *TYP files* specifying a mapping between program predicates and database tables in the DLV^{DB} syntax [38]. Besides the core functionality that basic text editors offer (like code line numbering, find/replace, undo/redo, copy/paste, etc.), *ASPIDE* offers other advanced functionalities, like: *Automatic completion*, *Dynamic code templates*, *Quick fix*, and *Refactoring*. Indeed, the system is able to complete (on request) predicate names, as well as variable names. Predicate names are both learned while writing, and extracted from the files belonging to the same project; variables are suggested by taking into account the rule we are currently writing. When several possible alternatives for completion are available the system shows a pop-up dialog. Moreover, the writing of repeated programming patterns (like transitive closure or disjunctive rules for guessing the search space) is assisted by advanced auto-completion with code templates, which can generate several rules at once according to a known pattern. Note that code templates can also be user defined by writing DLT [39] files. The refactoring tool allows one to modify in a guided way, among others, predicate names and variables (e.g., variable renaming in a rule is done by considering bindings of variables, so that variables/predicates/strings

occurring in other expressions remain unchanged). Reported errors or warnings can be automatically fixed by selecting (on request) one of the system's suggested quick fixes, which automatically change the affected part of code.
- *Outline navigation*. *ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files), and also provides a graphical support for building rules in the visual editor (see below).
- *Dynamic code checking and error highlighting*. Syntax errors and relevant conditions (like safety) are checked *while typing programs*: portions of code containing errors or warnings are immediately highlighted. Note that the checker considers the entire project, and warns the user by indicating e.g., that atoms with the same predicate name have different arity in several files. This condition is usually revealed only when programs divided in multiple files are run together.
- *Dependency graph*. The system is able to display several variants of the dependency graph associated to a program (e.g., depending on whether both positive and negative dependencies are considered).
- *Debugger and Profiler*. Semantic error detection as well as code optimization can be done by exploiting graphic tools. In particular, we developed a graphical user interface for embedding in *ASPIDE* the debugging tool *spock* [23] (we have also adapted spock for dealing with the syntax of the DLV system). Regarding the profiler, we have fully embedded the graphical interface presented in [40].
- *Unit Testing*. The user can define unit tests and verify the behavior of program units. The language for specifying unit tests, as well as the graphical tools of *ASPIDE* assisting the development of tests, are described in detail in the following sections.
- *Configuration of the execution*. This feature allows one to configure and manage input programs and execution options (called *run configurations*).
- *Presentation of results*. The output of the program (either answer sets, or query results) are visualized in a tabular representation or in a text-based console. The result of the execution can be also saved in text files for subsequent analysis.
- *Visual Editor*. The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules [41]. The user can switch, every time he needs, from the text editor to the visual one (and vice-versa) thanks to a reverse-engineering mechanism from text to graphical format.
- *Interaction with databases.* Interaction with external databases is useful in several applications (e.g., [12]). *ASPIDE* provides a fully graphical import/export tool that automatically generates mappings by following the DLV^{DB} TYP file specifications [38]. Text editing of TYP mappings is also assisted by syntax coloring and auto-completion. Database oriented applications can be run by setting DLV^{DB} as solver in a run configuration.

Interface Overview The user interface of *ASPIDE* is depicted in Figure 1. The most common operations can be quickly executed through a toolbar present in the upper part of the GUI (zone 1). From left to right there are buttons allowing to: save files, undo/redo, copy & paste, find & replace, switch between visual to text editor, run the solver/profiler/debugger. The main editing area (zone 4) is organized in a multi-tabbed panel possibly collecting several open files. On the left there is the explorer panel (zone 2) which allows one to browse the workspace; and the error console (zone 3). The explorer panel lists projects and files included in the workspace, while the error console organizes errors and warnings according to the project and files where they are localized. On the right, there are the outline panel (zone 5) and the sources panel (zone 6). The first shows an outline of the currently edited file, while the latter reports a list of the database sources connected with the current project. Note that, the layout of the system can be customized by the user, indeed panels can be moved and rearranged.

ASPIDE is written in Java and runs on the most diffused operating systems (Microsoft Windows, Linux, and Mac OS) and can connect to any database supporting Java DataBase Connectivity (JDBC).

3 A language for testing ASP programs

Software testing [42] is an activity aimed at evaluating the behavior of a program by verifying whether it produces the required output for a particular input. The goal of testing is not to provide means for estab-

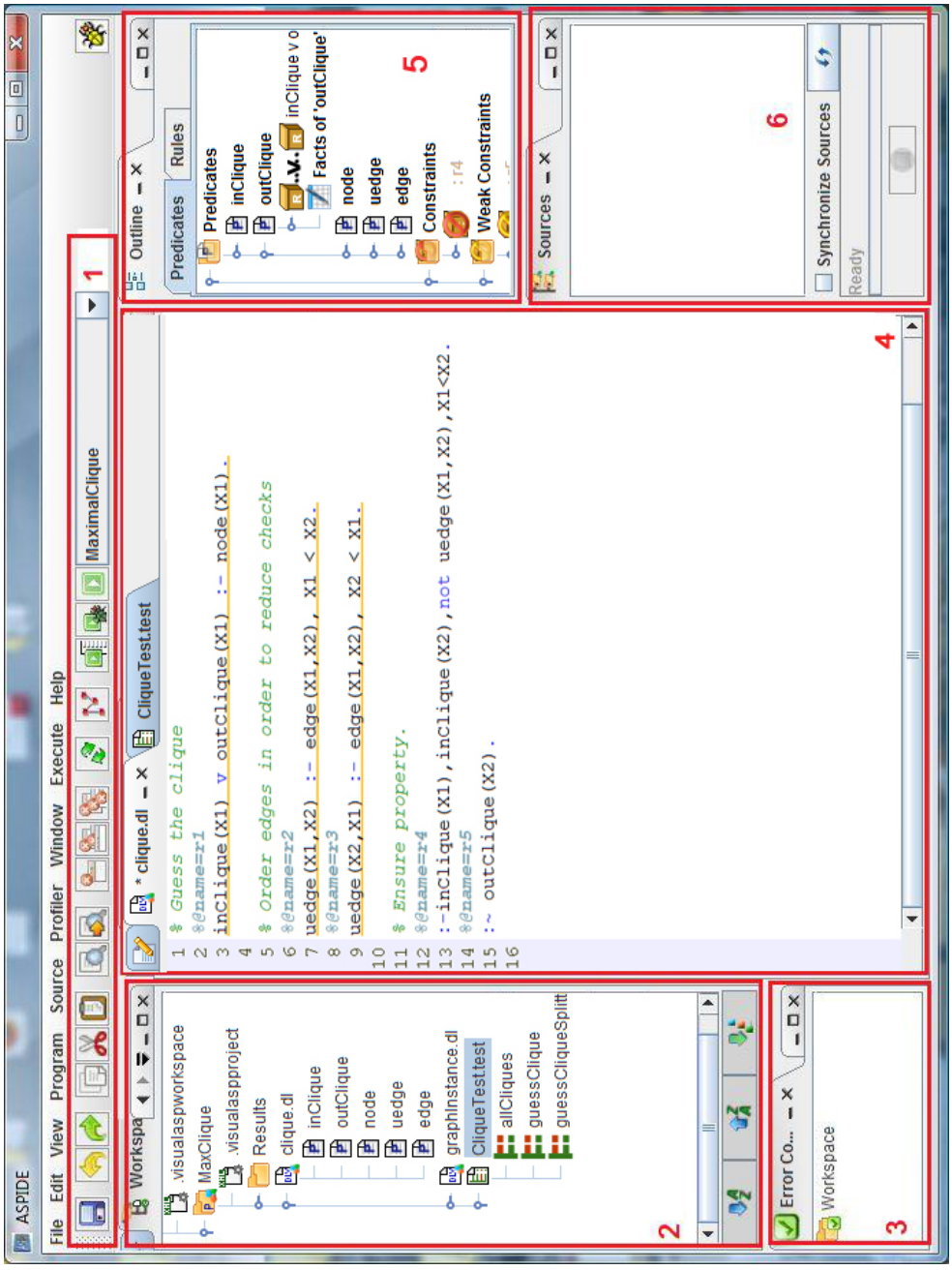


Fig. 1. The *ASPIDE* graphical user interface.

lishing whether the program is totally correct; conversely testing is a pragmatic and cheap way of finding errors by executing some test. A test case is the specification of some input I and corresponding expected outputs O. A test case fails when the outputs produced by running the program does not correspond to O, it passes otherwise.

One of the most diffused white-box[2] testing techniques is *unit testing*. The idea of unit testing is to assess an entire software by testing its subparts called *units* (and corresponding to small testable parts of a program). In a software implemented by using imperative object-oriented languages, unit testing corresponds to assessing separately portions of the code like class methods. The same idea can be applied to ASP, once the notion of unit is given. We intend as unit of an ASP programs P any subset of the rules of P corresponding to a splitting set [43] (actually the system exploits a generalization of the splitting theorem by Lifschitz and Turner [43] to the non-ground case [44]). In this way, the behavior of units can be verified (by avoiding unwanted behavioral changes due to cycles) both when they run isolated from the original program as well as when they are left immersed in (part of) the original program.

In the following, we present a pragmatic solution for testing ASP programs, which is a new language, inspired by the JUnit [33] framework, for specifying and running *unit tests*. The developer, given an ASP program, can select the rules composing a unit, specify one or more inputs, and assert a number of conditions on the expected output. The obtained test case specification can be run, and the assertions automatically verified by calling an ASP solver and checking its output. In particular, we allow three test execution modes:

- *Execution of selected rules*. The selected rules will be executed separated from the original program on the specified inputs.
- *Execution of split program*. The program corresponding to the splitting set containing the atoms of the selected rules is run and tested. In this way, the ”interface” between two splitting sets can be tested (e.g., one can assert some expected properties on the candidates produced by the guessing part of a program by excluding the effect of some constraints in the checking part).
- *Execution in the whole program*. The original program is run and specific assertions regarding predicates contained in the unit are checked. This corresponds to filtering test results on the atoms contained in the selected rules.

Testing Language. A test file can be written according to the following grammar:[3]

```
1 : invocation("invocationName" [ ,"solverPath", "options" ]?);
2 : [ [ input("program"); ] | [ inputFile("file"); ] ]*
3 : [
4 : testCaseName([ SELECTED_RULES | SPLIT_PROGRAM | PROGRAM ]?)
5 : {
6 : [newOptions("options");]?
7 : [ [ input("program"); ] | [ inputFile("file"); ] ]*
8 : [ [ excludeInput("program"); ]
9 : | [ excludeInputFile("file"); ] ]*
10 : [
11 : [ filter | pfilter | nfilter ]
12 : [ [ (predicateName [ ,predicateName ]* ) ]
13 : | [SELECTED_RULES] ] ;
14 : ]?
15 : [ selectRule(ruleName); ]*
16 : [ [ assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ); ]
17 : | [ assertBestModelCost(intcost [, intlevel ]? ); ] ]*
18 : }
19 : ]*
20 : [ [ assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ); ]
21 : | [ assertBestModelCost(intcost [, intlevel ]? ); ] ]*
```

A test file might contain a single test or a test suite (a set of tests) including several test cases. Each test case includes one or more assertions on the execution results.

The *invocation* statement (line 1) sets the global invocation settings, that apply to all tests specified in the same file (name, solver, and execution options). In the implementation, the invocation name might correspond to an *ASPIDE* run configuration, and the solver path and options are not mandatory.

[2] A test conceived for verifying some functionality of an application without knowing the code internals is said to be a black-box test. A test conceived for verifying the behavior of a specific part of a program is called white-box test. White box testing is an activity usually carried out by developers and is a key component of agile software development [42].

[3] Non-terminals are in bold face; token specifications are omitted for simplicity.

The user can specify one or more global inputs by writing some *input* and *inputFile* statements (line 2). The first kind of statement allows one for writing the input of the test in the form of ASP rules or simply facts; the second statement indicates a file that contains some input in ASP format.

A test case declaration (line 4) is composed by a name and an optional parameter that allows one to choose if the execution will be done on the entire program, on a subset of rules, or on the program corresponding to the splitting set containing the selected rules. The user can specify particular solver options (line 6), as well as certain inputs (line 7) which are valid in a given test case. Moreover, global inputs of the test suite can be excluded by exploiting *excludeInput* and *excludeInputFile* statements (lines 8 and 9). The optional statements *filter*, *pfilter* and *nfilter* (lines 11, 12, and 13) are used to filter out output predicates from the test results (i.e., specified predicate names are filtered out from the results when the assertion is executed).[4] The statement *selectRule* (line 15) allows one for selecting rules among the ones composing the global input program. A rule r to be selected must be identified by a name, which is expected to be specified in the input program in a comment appearing in the row immediately preceding r (see Figure 1). *ASPIDE* adds automatically the comments specifying rule names. If a set of selected rules does not belong to the same splitting set, the system has to print a warning indicating the problem.

The expected output of a test case is expressed in terms of assertion statements (lines 16/21). The possible assertions are:

- *assertTrue("atomList")*/*assertCautiouslyTrue("atomList")*. Asserts that all atoms of the atom list must be true in any answer sets;
- *assertBravelyTrue("atomList")*. Asserts that all atoms of the atom list must be true in at least one answer set;
- *assertTrueIn(number, "atomList")*. Asserts that all atoms of the atom list must be true in exactly *number* answer sets;
- *assertTrueInAtLeast(number, "atomList")*. Asserts that all atoms of the atom list must be true in at least *number* answer sets;
- *assertTrueInAtMost(number, "atomList")*. Asserts that all atoms of the atom list must be true in at most *number* answer sets;
- *assertConstraint(":-constraint.")*. Asserts that all answer sets must satisfy the specified constraint;
- *assertConstraintIn(number, ":-constraint.")*. Asserts that exactly *number* answer sets must satisfy the specified constraint;
- *assertConstraintInAtLeast(number, ":-constraint.")*. Asserts that at least *number* answer sets must satisfy the specified constraint;
- *assertConstraintInAtMost(number, ":-constraint.")*. Asserts that at most *number* answer sets must satisfy the specified constraint;
- *assertBestModelCost(intcost)* and *assertBestModelCost(intcost, intlevel)*. In case of execution of programs with weak constraints, they assert that the cost of the best model with level *intlevel* must be *intcost*;

together with the corresponding negative assertions: *assertFalse*, *assertCautiouslyFalse*, *assertBravelyFalse*, *assertFalseIn*, *assertFalseInAtLeast*, *assertFalseInAtMost*. The *atomList* specifies a list of atoms that can be ground or non-ground; in the case of non-ground atoms the assertion is true if some ground instance matches in some/all answer sets. Assertions can be global (line 20-21) or local to a single test (line 16-17).

In the following we report an example of test case.

Test case example. The maximum clique is a classical hard problem in graph theory requiring to find the largest clique (i.e., a complete subgraph of maximal size) in an undirected graph. Suppose that the graph *G* is specified by using facts over predicates *node* (unary) and *edge* (binary), then the program in Figure 1 solves the problem.

The disjunctive rule ($\mathtt{r}_1$) guesses a subset S of the nodes to be in the clique, while the rest of the program checks whether S constitutes a clique, and the weak constraint ($\mathtt{r}_5$) maximizes the size of S. Here,

[4] *pfilter* selects only positive literals and excludes the strongly negated ones, while *nfilter* has opposite behavior.

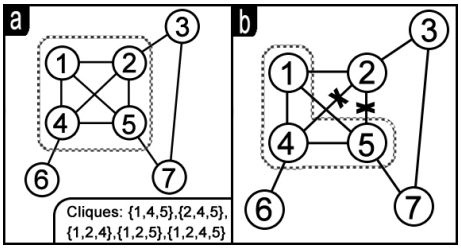


Fig. 2. Input graphs.

an auxiliary predicate *uedge* exploits an ordering for reducing the time spent in checking. Suppose that the encoding is stored in a file named *clique.dl*; and suppose also that the graph instance, composed by facts { *node(1). node(2). node(3). node(4). node(5). node(6). node(7). edge(1,2). edge(2,3). edge(2,4). edge(1,4). edge(1,5). edge(4,5). edge(2,5). edge(4,6). edge(5,7). edge(3,7).*}, is stored in the file named *graphInstance.dl* (the corresponding graph is depicted in Figure 2a). The following is a simple test suite specification for the above-reported ASP program:

```
invocation("MaximalClique", "/usr/bin/dlv", "");
inputFile("clique.dl");
inputFile("graphInstance.dl");
maximalClique()
{
assertBestModelCost(3);
}
constraintsOnCliques()
{
excludeInput(":~ outClique(X2).");
assertConstraintInAtLeast(1,":- not inClique(1), not inClique(4).");
assertConstraintIn(5,":- #count{ X1: inClique(X1) } < 3.");
}
checkNodeOrdering(SELECTED_RULES)
{
inputFile("graphInstance.dl");
selectRule("r2");
selectRule("r3");
assertFalse("uedge(2,1).");
}
guessClique(SPLIT_PROGRAM)
{
selectRule("r1");
assertFalseInAtMost(1,"inClique(X).");
assertBravelyTrue("inClique(X).");
}
```

Here, we first set the invocation parameters by indicating DLV as solver, then we specify the file to be tested *clique.dl* and the input file *graphInstance.dl*, by exploiting a global input statement; then, we add the test case *maximalClique*, in which we assert that the best model is expected to have a cost (i.e., the number of weak constraint violations corresponding to the vertexes out of the clique) of 3 (*assertBestModelCost(3)* in Figure 3).

In the second test case, named *constraintsOnCliques*, we require that (i) vertexes 1 and 4 belong to at least one clique, and (ii) for exactly five answer sets the size of the corresponding clique is greater than 2. (The weak constraint is removed to ensure the computation of all cliques by DLV.)

In the third test case, named *checkNodeOrdering*, we select rules r_2 and r_3, and we require to test selected rules in isolation, discarding all the other statements of the input. We are still interested in considering ground facts that are included locally (i.e., we include the file *graphInstance.dl*). In this case we assert that *uedge(2,1)* is false, since edges should be ordered by rules r_2 and r_3.

Test case *guessClique* is run in *SPLIT_PROGRAM* modality, which requires to test the subprogram containing all the rules belonging to the splitting set corresponding to the selection (i.e., {*inClique, outClique, node*}). In this test case the sub-program that we are testing is composed by the disjunctive rule and by the facts of predicate *node* only. Here we require that there is at most one answer set modeling the empty clique, and there is at least one answer set modeling a non-empty clique.

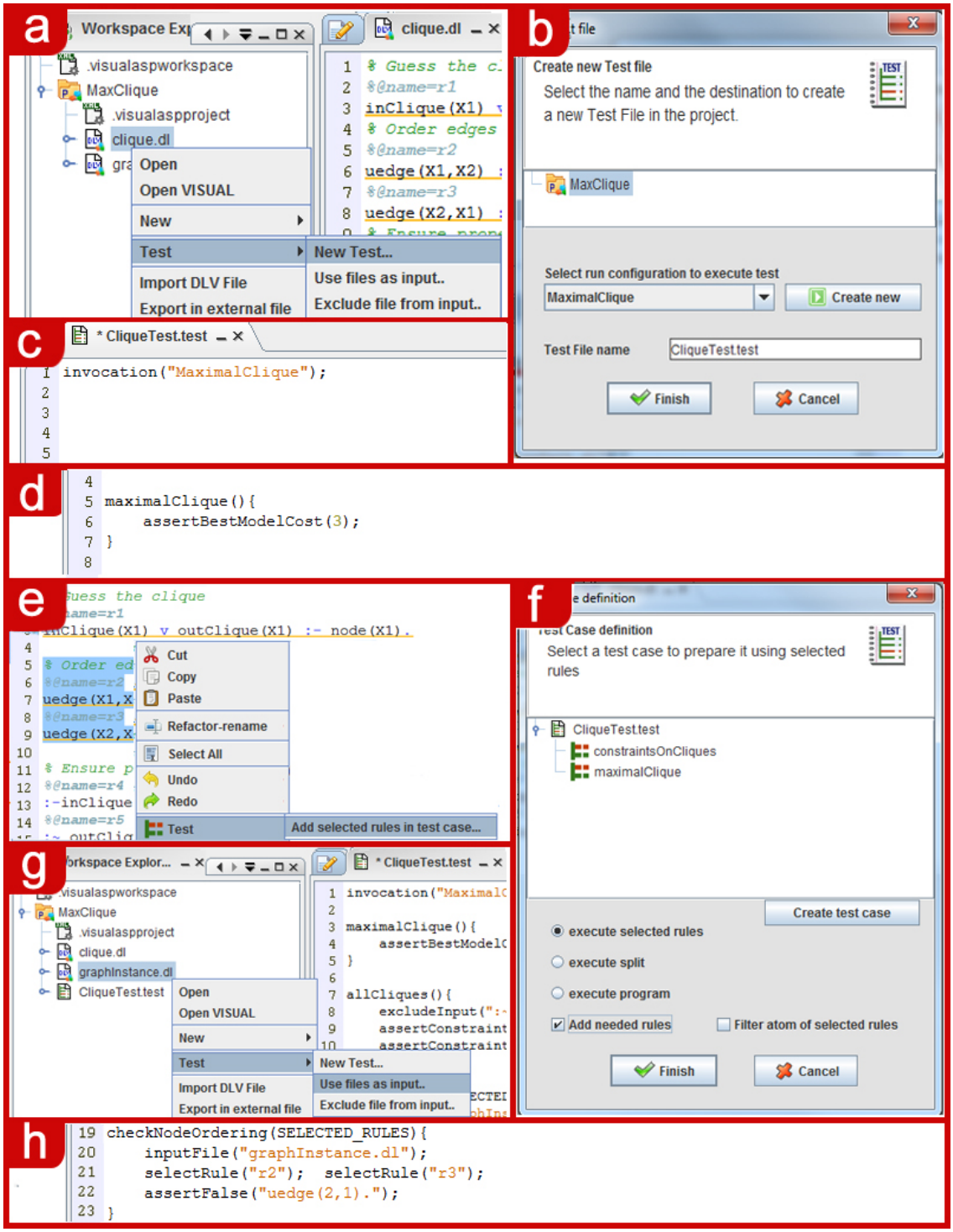


Fig. 3. Test case creation.

The test file described above can be created graphically and executed in *ASPIDE* as described in the following section.

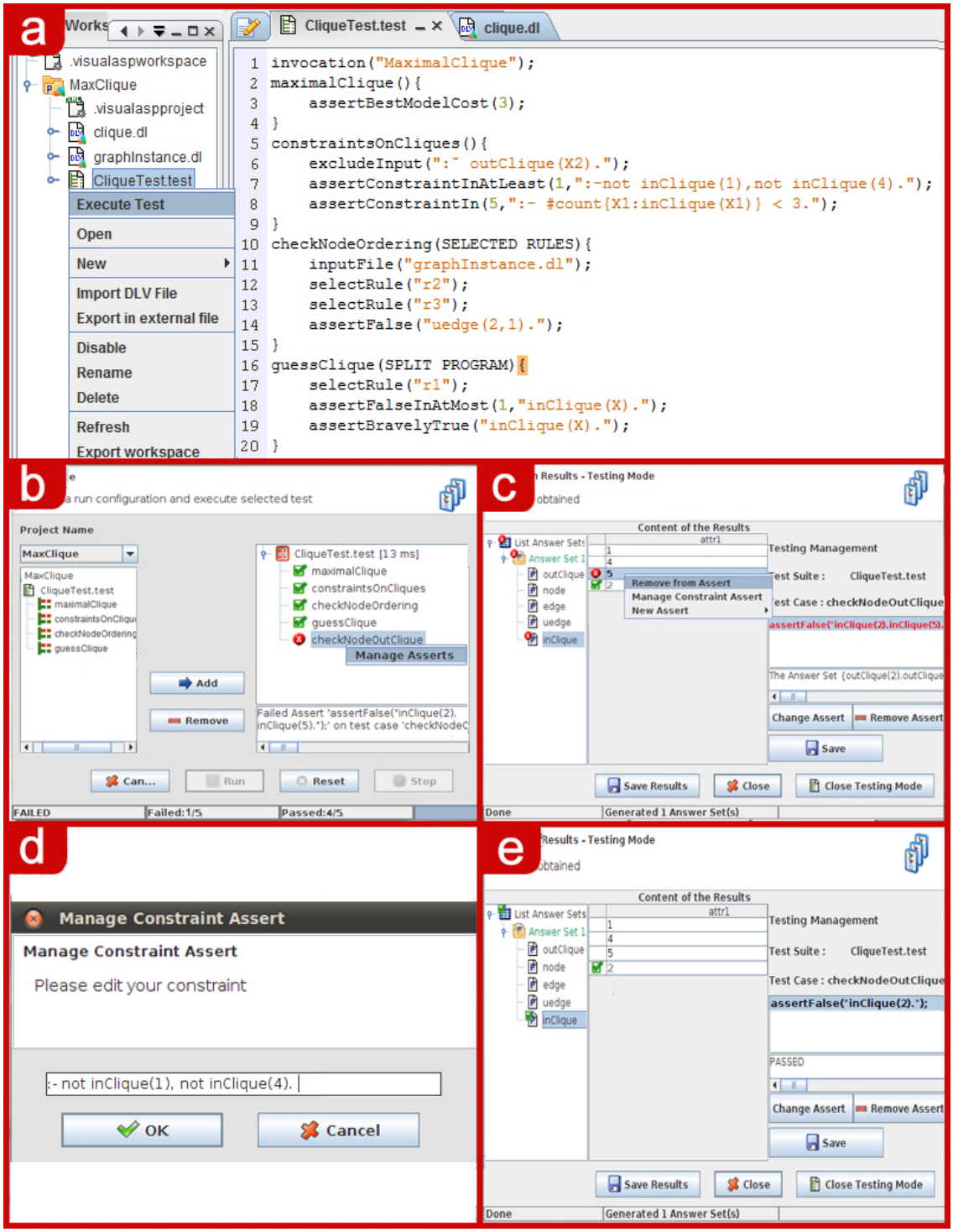


Fig. 4. Test case execution and assertion management.

4 Unit Testing in *ASPIDE*

In this section we describe the graphic tools implemented in *ASPIDE* conceived for developing and running test cases. Space constraints prevent us from providing a complete description of all the usage scenarios

and available commands. However, in order to have an idea about the capabilities of the testing interface of *ASPIDE*, we describe step by step how to implement the example illustrated in the previous section.

Suppose that we have created in *ASPIDE* a project named MaxClique, which contains the files *clique.dl* and *graphInstance.dl* (see Fig. 1) storing the encoding of the maximal clique problem and the graph instance presented in the previous section, respectively. Moreover we assume that both input files are included in a run configuration named *MaximalClique*, and we assume that the DLV system is the solver of choice in *MaximalClique*. Since the file that we want to test in our example is *clique.dl*, we select it in the *workspace explorer*, then we click the right button of the mouse and select *New Test* from the popup menu (Fig. 3a). The system shows the test creation dialog (Fig. 3b), which allows one for both setting the name of the test file and selecting a previously-defined run configuration (storing execution options and input files). By clicking on the *Finish* button, the new test file is created (see Fig. 3c) where a statement regarding the selected run configuration is added automatically. We add the first unit test (called *maximalClique*) by exploiting the text editor (see Fig. 3d), whereas we build the remaining ones (working on some selected rules) by exploiting the logic program editor. After opening the *clique.dl* file, we select rules r_2 and r_3 inside the text editor, we right-click on them and we select *Add selected rules in test case* from the menu item *Test* of the popup menu (fig. 3e). The system opens a dialog window where we indicate the test file in which we want to add the new test case (fig. 3f). We click on the *Create test case*; the system will ask for the name of the new test case and we write *guessClique*; after that, on the window, we select the option *execute selected rules* and click on the *Finish* button. The system will add the test case *guessClique* filled with the *selectRule* statements indicating the selected rules. To add project files as input of the test case, we select them from the *workspace explorer* and click on *Use file as input* in the menu item *Test* (fig. 3g). We complete the test case specification by adding the assertion, thus the test created up to now is shown in figure 3h. Following an analogous procedure we create the remaining test cases (see Fig. 4a). To execute our tests, we right-click on the test file and select *Execute Test*. The *Test Execution Dialog* appears and the results are shown to the programmer (see Fig. 4b). Failing tests are indicated by a red icon, while green icons indicate passing tests. At this point we add the following additional test:

```
checkNodeOutClique()
{
excludeInput("edge(2,4).edge(2,5).");
assertFalse("inClique(2). inClique(5).");
}
```

This additional test (purposely) fails, this can be easily seen by looking at Figure 2b; and the reason for this failure is indicated (see Fig. 4b) in the test execution dialog. In order to know which literals of the solution do not satisfy the assertion, we right-click on the failed test and select *Manage Asserts* from the menu. A dialog showing the outputs of the test appears where, in particular, predicates and literals matching correctly the assertions are marked in green, whereas the ones violating the assertion are marked in red (gray icons may appear to indicate missing literals which are expected to be in the solution). In our example, the assertion is *assertFalse("inClique(2). inClique(5).")*; however, in our instance, node 5 is contained in the maximal clique composed by nodes *1, 4, 5*; this is the reason for the failing test. Assertions can be modified graphically, and, in this case, we act directly on the result window (fig. 4c). We remove the node 5 from the assertion by selecting it; moreover we right-click on the instance of *inClique* that specifies the node 5 and we select *Remove from Assert*. The atom *node(5)* will be removed from the assertion and the window will be refreshed showing that the test is correctly executed (see fig. 4e). The same window can be used to manage constraint assertions; in particular, by clicking on *Manage Constraint Assert* of the popup menu, a window appears that allows the user to set/edit constraints (see fig. 4d).

5 Conclusion

This paper presents a pragmatic environment for testing ASP programs. In particular, we propose a new language, inspired by the JUnit [33] framework, for specifying and running *unit tests* on ASP programs. The testing language is general and suits both the DLV [15] and clasp [16] ASP dialects. The testing language has been implemented in *ASPIDE* together with some graphic tools for easing both the development of tests and the analysis of test execution (via DLV).

As far as future work is concerned, we plan to extend *ASPIDE* by improving/introducing additional dynamic editing instruments, and graphic tools like VIDEAS [45]. Moreover, we plan to further improve the testing tool by supporting (semi)automatic test case generation based on the structural testing techniques proposed in [32, 46].

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: ICLP'99 (1999) 23–37
3. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE **12**(5) (2000) 845–860
4. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. AI **175**(1) (2011) 278–298 Special Issue: John McCarthy's Legacy.
5. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
6. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
7. Balduccini, M., Gelfond, M., Watson, R., Nogeira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: LPNMR 2001 (LPNMR-01). LNCS 2173, (2001) 439–442
8. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence. Kluwer (2000) 257–279
9. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: LPNMR 2001 (LPNMR-01). LNCS 2173, (2001) 186–199
10. Franconi, E., Palma, A.L., Leone, N., Perri, S., Scarcello, F.: Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In: LPAR 2001. LNCS 2250, (2001) 561–578
11. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog Decision Support System for the Space Shuttle. In: Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001). LNCS 1990, (2001) 169–183
12. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, ACM Press (2005) 915–917
13. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
14. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In: LPNMR 2009. LNCS 5753, (2009) 591–597
15. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
16. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007,(2007) 386–392
17. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: KR 2008, Sydney, Australia, AAAI Press (2008) 422–432
18. Denecher, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming system competition. In: LPNMR'09. LNCS 5753, Potsdam, Germany, Berlin // Heidelberg (2009) 637–654
19. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The third answer set programming competition: Preliminary report of the system competition track. In: LPNMR11. LNCS 6645, (2011) 388–403
20. Dovier, A., Erdem, E.: Report on application session @lpnmr09 (2009) `http://www.cs.nmsu.edu/ALP/2010/03/report-on-application-session-lpnmr09/`.
21. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 86–100
22. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* Environment. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 101–115

23. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 71–85
24. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
25. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of the Sixth International Workshop on Automated Debugging, California, USA, ACM (2005)
26. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: On debugging non-ground answer-set programs. In: Proc. of the ICLP'10. (2010)
27. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: LPNMR'07. LNCS 4483, (2007) 31–43
28. De Vos, M., Schaub, T., eds.: SEA'07: Software Engineering for Answer Set Programming. In: . Volume 281., CEUR (2007) Online at `http://CEUR-WS.org/Vol-281/`.
29. De Vos, M., Schaub, T., eds.: SEA'09: Software Engineering for Answer Set Programming. In: . Volume 546., CEUR (2009) Online at `http://CEUR-WS.org/Vol-546/`.
30. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. Journal of Logic and Computation (2009)
31. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated Development Environment for Answer Set Programming. In: LPNMR'11, Vancouver, Canada, 2011. LNCS 6645, (May 2011) 317–330
32. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: ECAI 2010 Amsterdam, The Netherlands, The Netherlands, IOS Press (2010) 951–956
33. JUnit.org community: JUnit, Resources for Test Driven Development `http://www.junit.org/`.
34. Jack, O.: Software Testing for Conventional and Logic Programming. Walter de Gruyter & Co., Hawthorne, NJ, USA (1996)
35. Wielemaker, J.: Prolog Unit Tests `http://www.swi-prolog.org/pldoc/package/plunit.html`.
36. Cancinos, C.: Prolog Development Tools - ProDT `http://prodevtools.sourceforge.net`.
37. Calimeri, F., Ianni, G., Ricca, F.: The third answer set programming system competition (since 2011) `https://www.mat.unical.it/aspcomp2011/`.
38. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8** (2008) 129–165
39. Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: Answer Set Programming with Templates. In: ASP'03, Messina, Italy (2003) 239–252 Online at `http://CEUR-WS.org/Vol-78/`.
40. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A Visual Tracer for DLV. In: Proc. of SEA'09, Potsdam, Germany (2009)
41. Febbraro, O., Reale, K., Ricca, F.: A Visual Interface for Drawing ASP Programs. In: Proc. of CILC2010, Rende(CS), Italy (2010)
42. Sommerville, I.: Software Engineering. Addison-Wesley (2004)
43. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP'94, MIT Press (1994) 23–37
44. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artif. Intell. **172** (2008) 1495–1539
45. Oetsch, J., Pührer, J., Seidl, M., Tompits, H., Zwickl, P.: VIDEAS: A Development Tool for Answer-Set Programs based on Model-Driven Engineering Technology. In: LPNMR'11, Vancouver, Canada, 2011,. LNCS 6645, (May 2011) 382–387
46. Janhunen, T., Niemelä, I., Oetsch, J., Pührer J., Tompits, H.: Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison. In: LPNMR'11, Vancouver, Canada, 2011. LNCS 6645, (May 2011) 242–247

A Prototype of a Knowledge-based Programming Environment

Stef De Pooter, Johan Wittocx, and Marc Denecker

Department of Computer Science, K.U. Leuven

Abstract. In this paper we present a proposal for a knowledge-based programming environment. In such an environment, declarative background knowledge, procedures, and concrete data are represented in suitable languages and combined in a flexible manner. This leads to a highly declarative programming style. We illustrate our approach on an example and report about our prototype implementation.

1 Context

An obvious requirement for a powerful and flexible programming paradigm seems to be that within the paradigm different types of information can be expressed in suitable languages. However, most traditional programming paradigms and languages do not really have this property. In imperative languages, for example, non-executable background knowledge can not be described. The consequences become clear when we try to solve a scheduling problem in an imperative language: the background knowledge, the constraints that need to be satisfied by the schedule, gets mixed up with the algorithms. This makes adding new constraints and finding and modifying existing ones cumbersome.

On the other hand, most logic-based declarative programming paradigms lack the capacity to express procedures. Typically, they consist of a logic together with one specific type of inference. For example, Prolog uses Horn clause logic and does querying, in Description Logic the studied task is deduction, and Answer Set Programming and Constraint Programming make use of model generation. In such paradigms, whenever we try to perform a task that does not fit the inference mechanism at hand, the declarative aspect of the paradigm disappears. For example, when we try to solve a scheduling problem (which is a typical model-generation problem) in Prolog, then we need to represent the schedule as a term, say a list (rather than as a logical structure), and as a result the constraints do not really reside in the logic program, but will have to be expressed by clauses that iterate over a list [4]. Proving that a certain requirement is implied by another, is possible (in theory) for a theorem prover, but not in ASP. Etc.

To overcome these restrictions of existing paradigms, we propose a paradigm in which each component can be expressed in an appropriate language. We distinguish three components: procedures, (non-executable) background knowledge, and concrete data. For the first we need an imperative language, for the second an (expressive) logic, for the third a logical structure (which corresponds to a database). The connection between these components is mostly realized by various reasoning tasks, such as theorem proving, model generation, model checking, model revision, belief revision, constraint propagation, querying, datamining, visualization, etc.

The idea to support multiple forms of inference for the same logic or even for the same theories, was argued in [6]. Here it is argued that logic has a more flexible, multifunctional and therefore also more declarative role for problem solving than provided in many declarative programming paradigms, where typically one form of inference is central and theories are written to be used for this form of inference, sometimes even for a specific algorithm implementing this form of inference (such as PROLOG resolution). This view was therefore called the Knowledge Base System paradigm for declarative problem solving. The framework presented here is based on this view and goes beyond it in the sense that it offers a programming environment in which complex tasks can be programmed using multiple forms of inference and processing tools.

2 Overview of the language and system

To try out the above mentioned ideas in practice, we built a prototype interpreter that supports some basic reasoning tasks and a set of processing tools on high-level data such as vocabularies, structures and theories. In this section we will highlight various decisions in the design of our programming language and

interpreter. In the next section we will illustrate the usage of the language with an example. We named our language DECLIMP, which is an aggregation of "declarative" and "imperative".

2.1 Program structure

A DECLIMP program typically contains several blocks of code. Each block is either a procedure, a vocabulary (which is a list of sort, predicate and function names), a logic theory over vocabularies (which describes a piece of background knowledge using the relation and function names of its vocabulary), or a (possibly three-valued) structure over vocabularies. The latter represent databases over their vocabularies. To bring more structure into a program and to be able to work with multiple files, namespaces and include statements are provided.

Because vocabularies, logic theories and databases are not executable, and a program needs to be executed, control of a DECLIMP program is always in the hands of the procedures. Moreover, when a `main` procedure is available, the run of the program will start with the execution of this procedure. When there is no `main` procedure, the user can run commands in an interactive shell, after parsing the program.

In the next sections, we will describe the languages for the respective components in a DECLIMP program.

2.2 Knowledge representation language

For representing background knowledge we use an extended version of classical logic. A first advantage in using this language lies in the fact that classical logic is the best known and most studied logic. Also, classical logic has the important property that its informal semantics corresponds to its formal semantics. In other words, in classical logic the meaning of expressions[1] is intuitively clear. This is an important requirement in the design of a language that is accessible to a wider audience. Furthermore, there are already numerous declarative systems that use a language based on classical logic, or can easily be translated to it. Think of the languages of most theorem provers, various Description logics, and the language of model generators such as IDP [20, 8] and ENFRAGMO [14].

Research in the Knowledge Representation and Reasoning community has clearly shown that classical logic is in many ways insufficient. Aggregates and (recursive) definitions are well-known concepts that are common in the background knowledge of many applications, and which can generally not, or not in a concise and intuitively clear manner, be expressed in first-order logic. In DECLIMP we use an order-sorted version of first-order logic, extended with inductive definitions [5], aggregates [15], (partial) functions and arithmetic.

2.3 Structures

Structures in DECLIMP are written in a simple language that allows to enumerate all elements that belong to a sort and all tuples that belong to a relation or function. As an alternative to enumerating a relation, it is also possible to specify the relation in a procedural way, namely as all the tuples for which a given procedure returns 'true'. Furthermore, the interpretation of a function can be specified by a procedure, somewhat similar to "external procedure" in `DLV` [2].

As mentioned before, structures in DECLIMP are not necessarily two-valued. Three-valued structures are useful for representing incomplete information (which might be completed during the run of the program). To enumerate a three-valued relation (or function), two out of three of the following sets must be provided: tuples that certainly belong to the relation, tuples that certainly do not belong to the relation, and tuples for which it is unknown whether they belong to the relation or not. The third set can always be computed from the two given sets.

[1] We mean expressions that occur in practice, not artificially constructed sentences that do not really have meaning in real life.

2.4 Procedures

The imperative programming language in our prototype system is LUA [9]. The main reason for this choice is the fact that LUA is a lightweight scripting language and also because it has a good C++ API [10]. This facilitates on the one hand the compilation of programs written in DECLIMP and, on the other hand, the integration with the components of our DECLIMP interpreter, which is written in C++. When we do not take those reasons into account, any other imperative language is candidate.

In procedures, various reasoning methods on theories and structures can be called. Currently, the most important tasks supported by the DECLIMP-interpreter are the following:

Finite model expansion: Given a three-valued structure S and a theory T, find a completion of S to a two-valued structure that satisfies T. This is essentially a generalization of the reasoning task performed by ASP solvers, constraint programming systems, Alloy analyzers, etc. It is suitable for problems such as scheduling, planning and diagnosis. In our DECLIMP interpreter, model expansion is implemented by calls to the IDP system [20], which consists of the grounder GIDL [21] and solver MINISATID [11].

Finite model checking: Check whether a given two-valued structure is a model of a theory. This is an instance of model expansion and is implemented as such.

Constraint propagation: Deduce facts that must hold in all models of a given theory which complete a given three-valued structure. This is a useful mechanism in configuration systems [18] and for query answering in incomplete databases [3]. The propagation algorithm we implemented is described in [19].

Querying: Given an FO formula φ and a two-valued structure S, find all substitutions for free variables of φ that make φ true in S. The implementation of this mechanism makes use of Binary Decision Diagrams as described in [21].

Theorem proving: Given two FO theories T_1 and T_2, check whether $T_1 \models T_2$. This is implemented by calling a theorem prover provided by the user. In principle, any theorem prover that accepts TPTP [16] can be used.

Visualization: Show a visual representation of a given structure. We implement this by calling IDPDRAW, a tool for visualizing finite structures in which visual output is specified declaratively by definitions in our knowledge representation language or in ASP.

The values returned by the reasoning methods can be used in other reasoning methods and LUA-statements. We will illustrate this with an example in the next section.

3 Programming in DECLIMP

Say we want to write an application that allows players to solve sudoku puzzles. Such an application should be able to perform tasks such as generating puzzles, showing puzzles on the screen, checking whether solutions (player's choices) satisfy the sudoku rules, giving hints to the player, etc. In this application the different components we described before can clearly be distinguished: (1) the background knowledge consists of a logic theory containing the well-known sudoku constraints;

$$\forall r \forall n \exists! c : Sudoku(r,c) = n$$
$$\forall c \forall n \exists! r : Sudoku(r,c) = n$$
$$\forall b \forall n \exists! r \exists! c : InBlock(b,r,c) \wedge Sudoku(r,c) = n$$
$$\forall b \forall r \forall c : InBlock(b,r,c) \Leftrightarrow b = ((r-1)/3) * 3 + ((c-1)/3) + 1$$

(2) the data is stored in logical structures representing puzzles, and (partial and complete) solutions; and (3) the tasks we want it to perform, can be implemented using well-known inference methods.

Below we show (part of) a DECLIMP program. This code shows the use of an include statement and a namespace, and the declaration of a vocabulary `sudokuVoc` and a theory `sudokuTheory`, where the latter is simply an ASCII version of the theory shown above. Also note the `main` procedure at the bottom, which will be called when this program is passed to the interpreter.

```
#include "grid.idp"
```

```
namespace sudoku {

   vocabulary sudokuVoc {
      extern vocabulary grid::simpleGridVoc
      type Num isa nat
      type Block isa nat
      Sudoku(Row,Col) : Num
      InBlock(Block,Row,Col)
   }

   theory sudokuTheory : sudokuVoc {
      ! r n : ?1 c : Sudoku(r,c) = n.
      ! c n : ?1 r : Sudoku(r,c) = n.
      ! b n : ?1 r c : InBlock(b,r,c) & Sudoku(r,c) = n.
      ! r c b : InBlock(b,r,c) <=> b = ((r-1)/3)*3 + ((c-1)/3) + 1.
   }

   procedure solve(input) {
      return modelExpand(sudokuTheory,input)
   }

   procedure printSudoku(puzzle) {
      -- code for visualizing a sudoku puzzle.
   }

   procedure createSudoku() {
      math.randomseed(os.time())
      local puzzle = grid::makeEmptyGrid(9) -- defined in grid.idp

      stdoptions.nrmodels = 2
      local currsols = modelExpand(sudokuTheory,puzzle)
      while #currsols > 1 do
         repeat
            col = math.random(1,9)
            row = math.random(1,9)
            num = currsols[1][sudokuVoc::Sudoku](row,col)
         until num ~= currsols[2][sudokuVoc::Sudoku](row,col)
         makeTrue(puzzle[sudokuVoc::Sudoku].graph,{row,col,num})
         currsols = modelExpand(sudokuTheory,puzzle)
      end

      printSudoku(puzzle)
   }
}

procedure main() {
   sudoku::createSudoku()
}
```

Let us have a closer look at procedure `createSudoku` for creating sudoku puzzles. First it initializes an empty puzzle by instantiating a new logical structure. This is done by calling a procedure `makeSquare-Grid` which instantiates a structure with data about a generic grid of a certain size, and then adding domains for numbers and blocks particular for sudoku grids.

The second part of the procedure adds numbers to the grid until there is only one solution left for the puzzle. This is realized by performing model expansion (by calling `modelExpand`) to find two models of the theory that extend the given partially filled in puzzle. When two models are found, the algorithm selects a number that is unique for the first solution (that is, the number at the same position in the second solution is different) and is not yet present in the puzzle. When such an entry is found, it is added to the puzzle by

making the tuple {`row,col,num`} true in the interpretation of the function `Sudoku(Row,Col):Num`. Next, the procedure ask for two new models, and the process starts over. When only one model is found, the iteration stops, and procedure `printSudoku` is called to show the result on the screen using the visualization tool mentioned in the previous section.

4 Related work

There have been many proposals in the literature to combine procedural and declarative languages. A frequently occuring combination is that of a procedural language in which a program can post constraints expressed in an (often ad-hoc) declarative constraint language, while other primitives allow to call the constraint-solving process on the constraint store, express heuristics or call other processes, for example to edit or visualize output. Examples of systems with such languages are CPLEX [1], MOZART [17] and COMET [13]. These systems differ from DECLIMP in the sense that they offer only one kind of inference, namely constraint solving. A similar remark can be made about `CLP` and PROLOG systems with support for constraint propagation. Here the "procedural language" is the PROLOG language under its procedural semantics. In our system high-level concepts such as vocabularies, theories and structures are treated as first-class citizens that can be operated upon by arbitrary inference and processing tools, which offers more flexibility.

For another group of systems, control over execution of programs is in hands of one inference mechanism – or at least that inference is the main mechanism – and an integrated procedural language then allows users to stear some aspects of the inference mechanism, or for example format input and output, but do not allow to take over control. Examples of such systems are CLINGO [7] and ZINC [12]. The procedural languages in these systems have a more limited task then the one in DECLIMP. In DECLIMP the procedures are in control during execution, not just one of the inference mechanisms.

5 Conclusion

We have presented a knowledge-based programming environment, providing a declarative language for expressing background knowledge, an imperative programming language for writing procedures, and logical structures for expressing concrete data. The system also provides some state-of-the-art inference tools for performing various reasoning tasks.

We believe that a programming environment like the one proposed here overcomes some of the limitations of "single-programming-style" paradigms, by allowing a programmer to express the different types of information in software applications in appropriate languages. Making this explicit distinction in different types of information will increase readability, maintainability and reusability of programming code.

References

1. IBM ILOG CPLEX optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer.
2. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer Berlin / Heidelberg, 2008.
3. Marc Denecker, Álvaro Cortés Calabuig, Maurice Bruynooghe, and Ofer Arieli. Towards a logical reconstruction of a theory for locally closed databases. *ACM Transactions on Database Systems*, 35(3):22:1–22:60, 2010.
4. Marc Denecker, Danny De Schreye, and Yves Willems. Terms in Logic programs: a problem with their semantics and its effect on the programming methodology. *CCAI: Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 7(3-4):363–383, 1990.
5. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):Article 14, 2008.
6. Marc Denecker and Joost Vennekens. Building a knowledge base system for an integration of logic programming and classical logic. In María García de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 71–76. Springer, 2008.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. http://downloads.sourceforge.net/potassco/guide.pdf, 2010.

8. The IDP system. http://dtai.cs.kuleuven.be/krr/software.html.
9. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
10. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Queue*, 9:20:20–20:29, May 2011.
11. Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *LNCS*, pages 211–224. Springer, 2008.
12. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
13. Laurent Michel and Pascal Van Hentenryck. The Comet programming language and system. In Peter van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 881–881. Springer, 2005.
14. David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.
15. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.
16. Geoff Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
17. Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.
18. Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 141–148. ACM, 2009.
19. Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for extended first-order logic. *CoRR*, abs/1008.2121, 2010.
20. Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.
21. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.

A Visual Entity-Relationship Model for Constraint-Based University Timetabling

Islam Abdelraouf, Slim Abdennadher, Carmen Gervet

Department of Computer Science, German University in Cairo
[islam.abdelraouf, slim.abdennadher, carmen.gervet]@guc.edu.eg
http://met.guc.edu.eg

Abstract. University timetabling (UTT) is a complex problem due to its combinatorial nature but also the type of constraints involved. The holy grail of (constraint) programming: "the user states the problem the program solves it" remains a challenge since solution quality is tightly coupled with deriving "effective models", best handled by technology experts. In this paper, focusing on the field of university timetabling, we introduce a visual graphic communication tool that lets the user specify her problem in an abstract manner, using a visual entity-relationship model. The entities are nodes of mainly two types: resource nodes (lecturers, assistants, student groups) and events nodes (lectures, lab sessions, tutorials). The links between the nodes signify a desired relationship between them. The visual modeling abstraction focuses on the nature of the entities and their relationships and abstracts from an actual constraint model.

1 Introduction

University timetabling (UTT) is a complex problem due to its combinatorial nature but also the type of constraints involved. Several approaches have been proposed to solve timetabling problems for specific instances using several approaches, e.g. [5, 3, 1].
One of the shortcomings of the current constraint-based systems is the existence of modeling tools/languages to express constraint problems. Several approaches have been proposed to overcome these problems. A lot of work has been invested to propose languages that allow the specification of a combinatorial problem at a high level of abstraction, e.g. [2]. Alternatively, visual modeling languages have been proposed to generate constraint programs from visual drawings, e.g. [4]. In general, the visual drawings correspond to constraint graphs where the nodes describe the variables of the problem with their associated domains and the edges correspond to the constraints between each pair of variables. The constraint graphs provide a visual counterpart to constraint satisfaction problems. However, in practice they are intractable for real world problems even if abstraction is introduced into the constraint graph.
In this work, we propose an orthogonal approach where the user models the constraint problem visually by drawing a graph that defines the available resources and the tasks to be scheduled. The graph does not describe the constraints explicitly. It consists of nodes and links, where the nodes are mainly of two types: resource nodes (lecturers, assistants, student groups) and events nodes (lectures, lab sessions, tutorials). The links describe relationships between the nodes. Depending on the type of node, the semantics of the links is determined.
This approach enjoys three main properties: 1) the problem is stated at a high level in a constraint and variable free manner, 2) online preliminary consistency checks of proposed links are possible thanks the rich semantic carried by the nodes, 3) a compilation into an effective constraint programming model including global constraints is performed using the properties of the nodes and specified links.
Our system was built in Java and compiled into SICStus Prolog. Tests were run to build a complete timetable for the German University in Cairo (GUC) including over 200 events to be scheduled and over 400 resources in 3 minutes.
This paper is organized as follows. In Section 2, we present the GUC timetabling problem. In Section 3, we describe how the problem can be modeled as a constraint satisfaction problem. Section 4 presents the visual graphic communication tool for generic timetabling problems. In Section 5, we address additional problem components that are particular for the GUC. In Section 6, we discuss the scalability and the modularity of the approach. Finally, we conclude with a summary and directions for future work.

2 GUC Timetabling

The GUC consists of four faculties. Each faculty offers a set of majors. Currently at the GUC, there are 140 courses offered and 6500 students registered for which course timetables should be generated every semester. There are 200 staff members available for teaching. Each faculty offers a set of majors. For every major, there is a set of associated courses. Faculties do not have separate buildings, therefore all courses from all faculties should be scheduled taking into consideration shared room resources.
Students registered to a major are distributed among groups for lectures (lecture groups) and groups for tutorials or labs (study groups). A study group consists of maximum 25 students. In each semester, study groups are assigned to courses according to their corresponding curricula and semester. Due to the large number of students in a faculty and lecture hall capacities, all study groups cannot attend the same lecture at the same time. Therefore, sets of study groups are usually assigned to more than one lecture group. For example, if there are 27 groups studying Mathematics, then 3 lecture groups will be formed.
The timetable at the GUC spans a week starting from Saturday to Thursday. A day is divided into five time slots, where a time slot corresponds to 90 minutes. An event can take place in a time slot. This can be either a lecture, tutorial, or lab session and it is given by either a lecturer or a teaching assistant. Lectures are given by lecturers and tutorial and lab sessions are given by teaching assistants (TA). In normal cases, lectures take place in lecture halls, tutorials in exercise rooms and lab sessions take place in specialized laboratories depending on the requirements of a course. In summary, an event is given by a lecturer or a teaching assistant during a time slot in a day to a specific group using a specific room resource. This relationship is represented by a timetable for all events provided that hard constraints are not violated. These constraints cannot be violated and are considered to be of great necessity to the university operation. The timetable also tries to satisfy other constraints which are not very important or critical. Such constraints are known as soft constraints that should be satisfied but may be violated. For example, these constraints can come in form of wishes from various academic staff.
Some courses require specialized laboratories or rooms for their tutorials and lab sessions. For example, for some language courses a special laboratory with audio and video equipment is required. The availability of required room resources must be taken into consideration while scheduling. Some lecturers have specific requirements on session precedences. For example, in a computer science introductory course a lecturer might want to schedule tutorials before lab sessions.
Furthermore, some constraints should be taken into consideration to improve the quality of education. One of the constraints requires that a certain day should have an activity slot for students, and a slot where all university academics can meet. For those slots no sessions should be scheduled. A study group should avoid spending a full day at the university. In other words, the maximum number of slots that a group can attend per day is 4. Therefore, if a group starts its day on the first slot, then it should end its day at most on the fourth slot. Furthermore, though the university runs for 6 days a week each study group must have at least two days off which means that a study group can only have sessions for five days a week.
A certain number of academics are assigned to a course at the beginning of a semester. Teaching assistants are assigned to one course at a time. For courses involving lab and tutorial sessions, a teaching assistant can be assigned to both or just one of them. This should be taken into consideration when scheduling to avoid a possible clash. Furthermore, the total numbers of TAs assigned to a session should not exceed the maximum number of assistants assigned to the corresponding course at any time. A lecturer can be assigned to more than one course. This should be considered when scheduling in order to avoid a possible overlap. Academics assigned to courses have a total number of working and research hours per day and days off that need to be considered when scheduling. Courses should be scheduled in a manner such that the number of session hours given by an academic should not exceed his or her teaching load taking into consideration days off.

3 Modeling UTT as a Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is a tuple (X, D, C) where X is a set of variables $\{x_1, ..., x_n\}$, $D = \{D_1, ..., D_n\}$ a set of associated domains for all $x_i \in X$, and C a set of constraints [6].
In the CSP model of the UTT, we define the variables **E** (possibly subscripted) to be lecture slots, tutorial slots, and lab slots taken by groups. Since we have 5 slots in a day and 6 days in a week, the domain is

$[0..29]$. Slot 0 would correspond to the first slot on Saturday (first day of the week in Egypt), and slot 22 would correspond to the third slot on Wednesday. Consequently, slots from 0 to 4 correspond to Saturday, 5 to 9 correspond to Sunday, etc. The goal is to find an allocation of one slot per variable such that the problem resource and scheduling constraints hold.
The generic UTT constraints are of three kinds: 1) temporal constraints requiring precedences among events ($\mathbf{E_i} \leq \mathbf{E_j}$), 2) resource constraints requiring events sharing some resources not to overlap in time, 3) and university specific constraints such as preferences for lecturers and teaching Assistants.
Note that resource constraints can take two forms in the timetabling problem:

1. when different events share a single resource, like a number of lectures taught by one lecturer, in this case none of those events can overlap. They can be modeled using inequality ($\mathbf{E_i} \neq \mathbf{E_j}$) among events associated with the same resource or the `all_different`($[\mathbf{E_i}, \mathbf{E_j}]$) global constraint,
2. when a set of events share multiple resources. This generalizes the previous form and it occurs in the UTT problem for instance when a set of events require the same type of rooms (halls or labs) and there is a limited amount of them. A global constraint can be used to enforce this which is the `cumulative/2` constraint. In SICStus Prolog, the `cumulative` constraint takes a list of `task` predicates defined by their start time, duration, end time, resource usage and identifier. Its second argument is a list containing the amount of resource available. It will be used in our code generation.

The last type of constraints applied are university specific constraints which for the GUC relate to the extra day off (not bound to Saturday) and the maximum allowed slots per day for study groups. A day off would mean a day with no events scheduled for that lecturer or TA. This can be expressed as follows if we consider Saturday as the day-off for a given lecturer. All the lectures relative to a lecturer(we denote by L_i the list of events associated with lecturer i) must start after time slot 4:

$$\forall_j \mathbf{E_j} \in L_i \Rightarrow \mathbf{E_j} > 4$$

In general, timetabling is an over-constrained problem. Thus, the main aim is to find the best solution that satisfies all hard constraints and as many soft constraints as possible. For the German University in Cairo, different lecturers have different requirements in terms of days off and time slots they would like to have free. Thus, the best solution for the German University In Cairo is the one that satisfies the largest number of the lecturers' wishes. These soft constraints are modeled by associating a flag with each wish that determines whether the wish is satisfied or not. For example, the constraint that a lecturer i would like to have the first slot on Saturday free can be modeled as follows:

$$0 \notin L_i \Longrightarrow P_k = 1$$

where P_k is the flag associated with the k^{th} wish.
The best solution for the GUC would be the one that maximizes the number of satisfied wishes, i.e., the solution maximizes a cost function that is defined as follows:

$$SCORE = \sum_{n=1}^{n=m} P_n$$

where m is total number of wishes .

4 From a visual graph to a CSP program

Our goal is to offer the end-user a tool that would allow her to specify the problem as she sees it, while enabling us to derive from it an effective constraint model. To do so we chose the concept of graph as a natural vehicle of connections or relationships between multiple entities or nodes. In this section we define the components that define a graphical specification elaborating on the semantic of the nodes which is carried visually and internally as a typed data structure. It applies to the generic components of UTT problems. We then give the modular structure of the tool and show how a graphical specification is compiled into a SICStus program. We finally develop the specification of additional problem components that are particular to a given university and are specified using other elements of our tool.

4.1 Visual graph components

The graph is not a constraint graph in the sense that nodes are not variables and edges constraints. The visual structure of a graph has been chosen to ease the abstract specification of the problem. Its visual nature helps convey the problem structure to third party (faculty deans, administration,etc). It is structured in a way that aids in both problem specification and future compilation into a CP model. Elements of the timetabling problem are available resources (lecturers, teaching assistants, study groups), courses, and different course events. The graph is composed of nodes and undirected links. As the user constructs her visual graph to specify her problem, certain information must be held at the node to aid in the dynamic CP model generation. In particular the nodes specify different components of a UTT. Let us first describe the different types the nodes can denote.

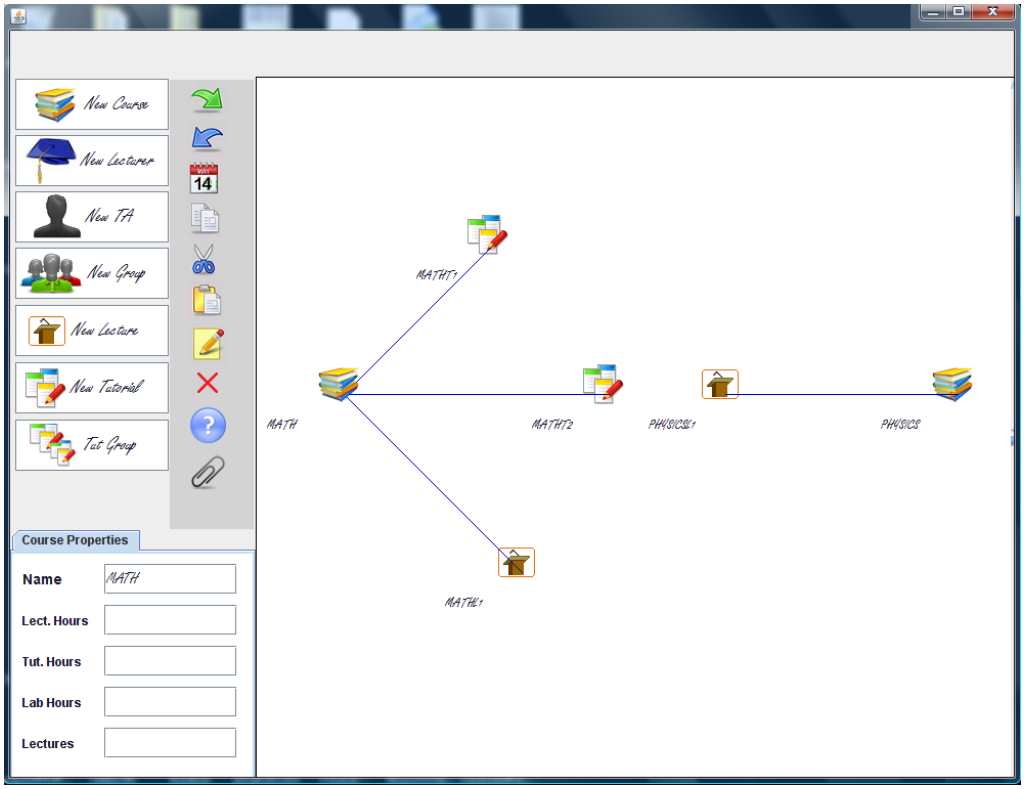


Fig. 1. Created graph with two courses and 4 events

Nodes. Nodes can be of three types:

1. A resource. These are lecturers, study groups or teaching assistants.
2. An event. These are lectures, tutorials or lab sessions.
3. Meta-events. These are the courses. A course is often composed of a lecture and a tutorial and possibly some lab sessions. Thus, the end-user can define a course and associate with it the relevant events it involves. This type of node does not appear in the generated CSP model, it is used to ease the visual specification of the problem.

Each node instance corresponds to a specific icon, carrying out its semantics from the end-user viewpoint. To illustrate drawing facilities of the system, we give a snapshot of a small graph being created. The end-user defined two courses (meta-events), Math and Physics, connected to 3 and 1 events, respectively. The Math course comes with 2 tutorial sessions and one lecture, whereas the Physics course comes so far with

only one lecture. In Fig. 1, we present the system with the drawing graph components. On the left hand side, the node icons are available. On the right hand side, the drawing pad, where the graph is created, is available. Below the node icons, the node properties to be added once a node has been inserted into the graph tab is available.
The icon indicates the type of a node and properties associated with it. For instance, lectures and tutorials require certain room size; lectures require lecture halls while tutorials require classrooms. Representing this on the diagram would require creating an icon or a node for each room type and connecting this node to all events that require this type of node. Since all events require a certain type of room, displaying the rooms as an extra icon would increase the number of edges and make the graph harder to trace. Instead of connecting events that share a certain type of rooms to a single resource node, all events requiring the same room resource have similar icons (lectures have the same icon which is different from that of tutorials and labs). Additional properties are filled in for each created node in the property tab.

Links. Links between nodes are undirected connections, signifying a relationship between two nodes (ie. a resource will carry out an event, or a course is defined by one lecture and 3 tutorials). The approach is user-friendly, but could be prone to specification errors. Thus while the end-user does not deal with node types explicitly, the system performs a dynamic checking at link creation to ensure that resource-resource links can not be validated, and resource-event links are only created if the event suits the type of the resource (e.g. a lecturer can only teach lectures no tutorials or labs). This is described below.

4.2 Graph semantic check

During the creation of the graph, we perform a dynamic semantic check that prevents incorrect link requests from being drawn between two nodes. While the user can add any number of nodes, certain connections would not make sense.
The semantic check is performed as the end-user attempts to connect two previously created nodes. When the user attempts such a connection, a request is sent that ensures that:

- A course can only be connected to lectures, tutorials or labs nodes.
- A lecture node can only be connected to one lecturer, one course and study groups.
- A tutorial or lab node can only be connected to one TA, one course and one group.
- A group node can only be connected to event nodes (lectures, tutorials or labs).
- A lecturer node can only be connected to lectures nodes.
- A TA node can only be connected to tutorials or labs nodes.

Any other connection request is ignored and no line can be drawn between the desired nodes.

Node implementation. At the implementation level, each resource node is associated with a dynamic list of events this resource node is currently connected to. For instance a lecturer node would be linked to a list of lectures, the person is assigned to. Each event node (Lecture, tutorial, lab) is linked with two flags: one flag stating whether this node is connected to its corresponding event (lecture to a lecturer, lab to a TA, etc). This flag is primarily used to prevent a single event node from being connected to multiple resource node of the suitable type. The second flag states whether the event node is connected to a course and is used to prevent a single event from being connected to multiple courses. A static information is also attached to each resource node which is the constraint predicate the list will be applied to.

4.3 Generic CP code generation

This is a core component of the system. The main objective is to generate an efficient CP code from the visual graph and node properties that goes beyond binary constraints and includes global constraints. Our code generation module is based on the following observations: 1) links involving resource nodes drive the model. Thus when a resource node is created in the graph we create a dynamic list in the CSP model that will contain all the events connected to it, 2) all the established connections are semantically viable because the dynamic checks at link creation have been carried out beforehand.
Since the constraints are derived from the relations between nodes then a node must be aware of all its direct neighbors. In other words, each node contains a list of nodes representing its direct neighbors. Each node has a type from which we know the type of constraints to be applied.

The alldifferent constraint . Consider the graph of Fig. 1, there is no resource node involved at this stage, thus no constraints are generated yet. Adding the lecturer as illustrated in Fig 2. will constrain events that share a common resource and trigger the code generation.

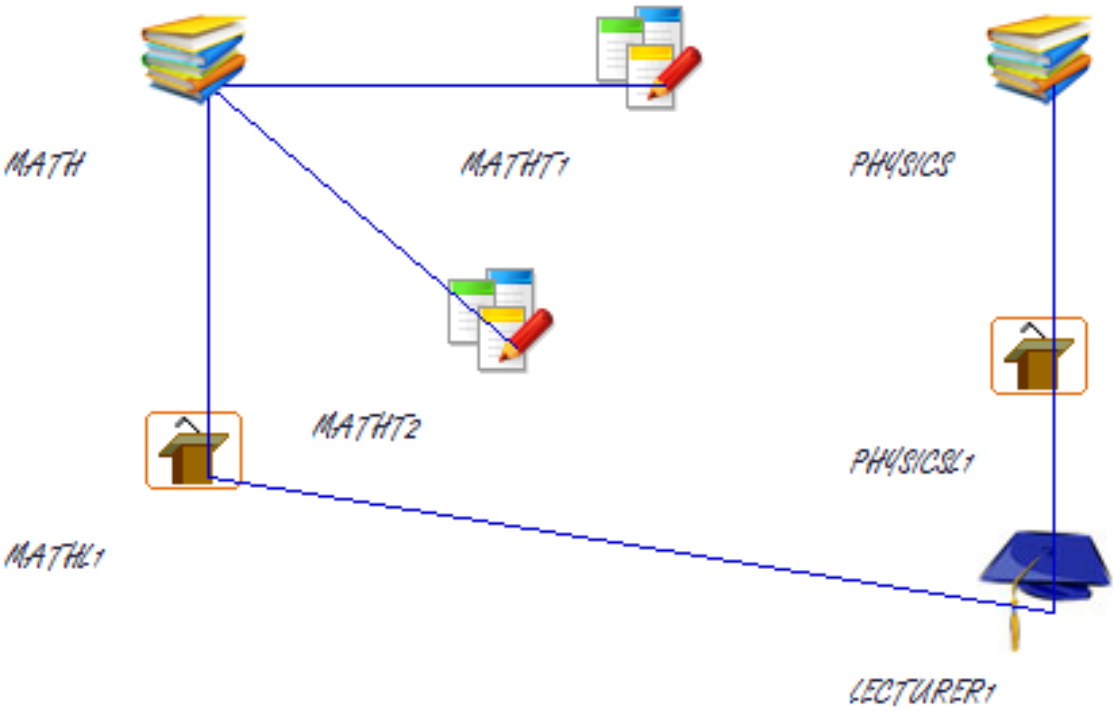


Fig. 2. Two courses talked by one lecturer

This establishes a resource dependency link between event nodes. From a constraint point of view, this implies that the two lecture events (MATH and PHYSICS) share a common resource, thus can not occur simultaneously. The system generates a list of event variables (corresponding to the events connected to a resource node) and constrains all the variables in this list to be pairwise distinct. The most efficient constraint available to this date in CP systems to enforce this restriction is the `all_different/1` constraint. It takes as input a list of event variables that cannot share a value. In our case, the domain of event variables is the time slot at which it can occur. Note that the initial domains are defined as the system is initialized, since the user will enter the number of slots per day and the number of days per week. In the case of the GUC we have $5 \times 6 = 30$ (with domains $[0..29]$). The following code block will be generated as the lecturer node is connected:

```
LECTURER1 = [MATHL1, PHYSICSL1],
all_different(LECTURER1),
```

Now if the user adds resource nodes relative to the study groups as illustrated in Fig. 3, two blocks of code will be generated relative to these resources.
In this case, the code block generated on creation of the node `GROUP1` and its relative links, is:

```
GROUP1 = [MATHL1, MATHT1, MATHT2],
all_different(GROUP1),
```

and the code generated by `GROUP2` will be:

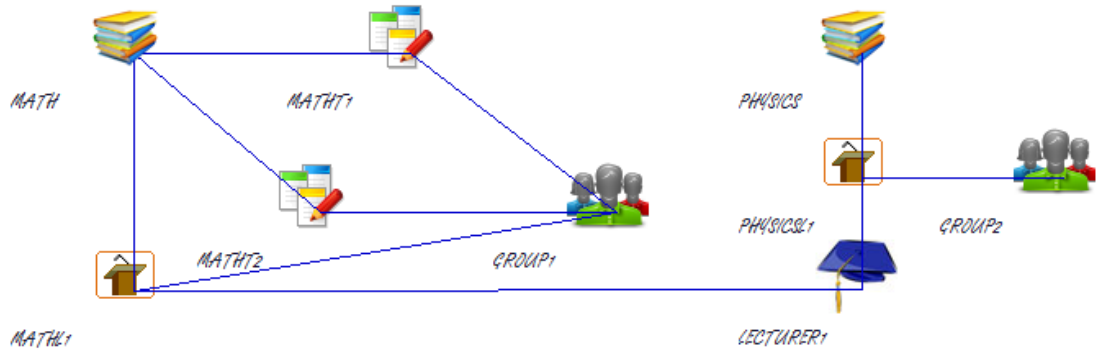


Fig. 3. Addition of study groups

```
GROUP2 = [PHYSICSL1],
all_different(GROUP2),
```

Note that the code is updated dynamically, which means that as a new event is connected to a given resource node, the list of events associated to this node is updated and so is the relative code block.

The cumulative constraint . Lectures and tutorials require specific rooms: lectures require lecture halls while tutorials require classrooms. As mentioned before, all events requiring the same room type (or resource) have similar icons. The most effective way to constrain events that share a finite number of resources is the global `cumulative/2` constraint. This constraint takes as input a set of event variables with possible start dates as domain, and number of possible resources to share. It ensures that the number of overlapping events requiring a common room type should not exceed the number of rooms available of that type. We generate the code block using the `cumulative` constraint for each room type existing in the university and each event is included in its corresponding `cumulative` constraint according to its type. The program generation is done by concatenating blocks of code generated by node with the `cumulative` constraints and the domain constraint. The block code generated is as follows:

```
domain([MATHT1E0, MATHT2E1], 1, 4),
%% each event last one slot and consumes 2 rooms,
%% there are atmost 2 rooms available in this example
cumulative([task(MATHT1, 1, MATHT1E0, 2, 0),
task(MATHT2, 1, MATHT2E1,  2, 1)],
[limit(2), global(true)]),

domain([PHYSICSL1E0, MATHL1E1], 1, 4),
cumulative([task(PHYSICSL1, 1, PHYSICSL1E0, 2, 0),
task(MATHL1, 1, MATHL1E1, 2, 1)],
[limit(2), global(true)]),
```

The system with the draw tab allows the user to visualize the generated code as illustrated in Fig 4.

The window is divided into three major sections; the first contains the graph, below it are two other windows, one appears whenever the user clicks on a node which allows the user to provide simple information that is node specific like the node name. The third window shows the user the generated SICStus program. The model specified and generated so far is generic.

However, a UTT problem deals with university specific constraints and preferences and these should be part of the visualization and the CP model. We present two other features of our system, namely the time tab, and the preference tab.

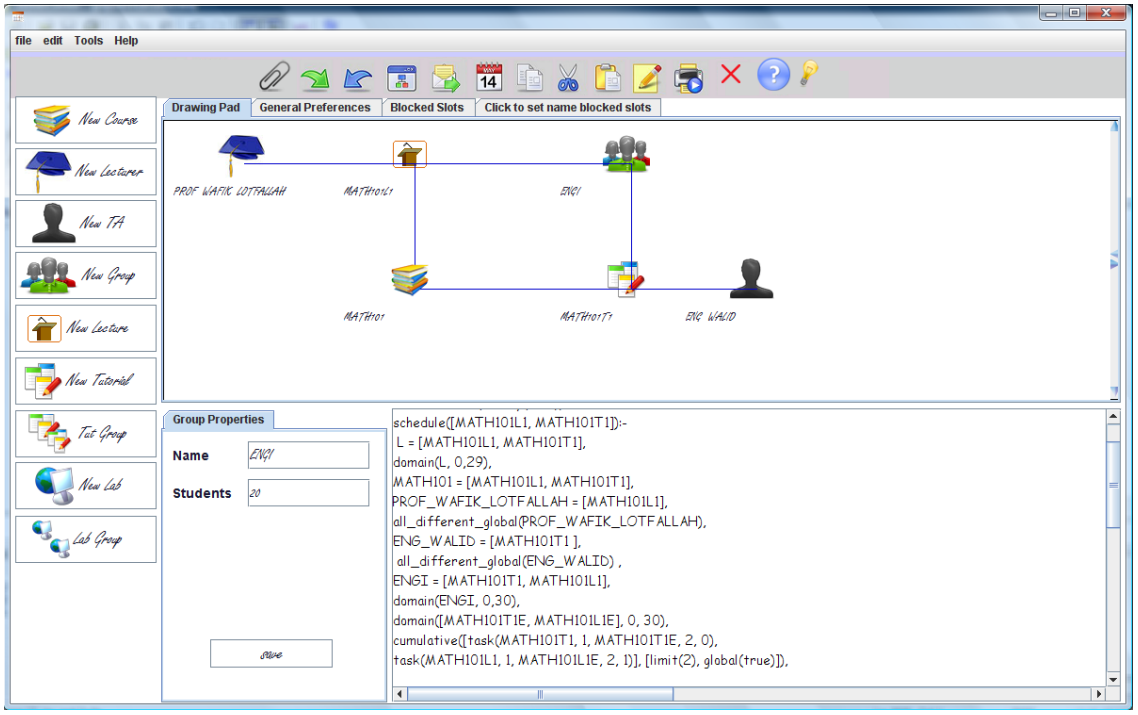


Fig. 4. The system with the draw tab

5 University specific constraints

The use of specific icons indicates the type of room an event node is related to. This eliminated the need for a room resource node. However, to ensure that there is enough room resources available, we must know the capacity of each room resource, and we must know the number of rooms of that type. These informations can not be acquired from the graph. Therefore we provide complementary means for the user to provide additional information. The user can choose between different tabs. The first tab described previously deals with the graph construction. We now present the preference tab and the time tab.

5.1 Dealing with preferences

The second tab (Fig. 5) enables the user to tailor the problem to a specific university and set preferences (room availability, extra day-off considered or not, full day for student groups or not).
All node types can access the preference tab and thus information can be exchanged. If the day off for a study group is selected a corresponding code block will be generated. The following block of code will be generated as for LECTURER1. It is a hard constraint then ensures that only one extra day-off is allowed per resource node.
The `count_interval` is a user defined constraint that constrains the number of events per resource (list `LECTURER1`) that fall in the time interval (eg. [0,2]) to be `LECTURER1DOC`.

```
count_interval(0, 2, LECTURER1, LECTURER1D0C),
LECTURER1D0C #\= 0 #<=> LECTURER1D0,
count_interval(3, 5, LECTURER1, LECTURER1D1C),
LECTURER1D1C #\= 0 #<=> LECTURER1D1,
count_interval(6, 8, LECTURER1, LECTURER1D2C),
LECTURER1D2C #\= 0 #<=> LECTURER1D2,
LECTURER1D2 + LECTURER1D1 + LECTURER1D0 #< 3,
```

Also, according to the GUC policy, a student should not have a full day which means a student having a class on the first time slot of a day can not have a class on the last time slot of the same day. Ensuring this

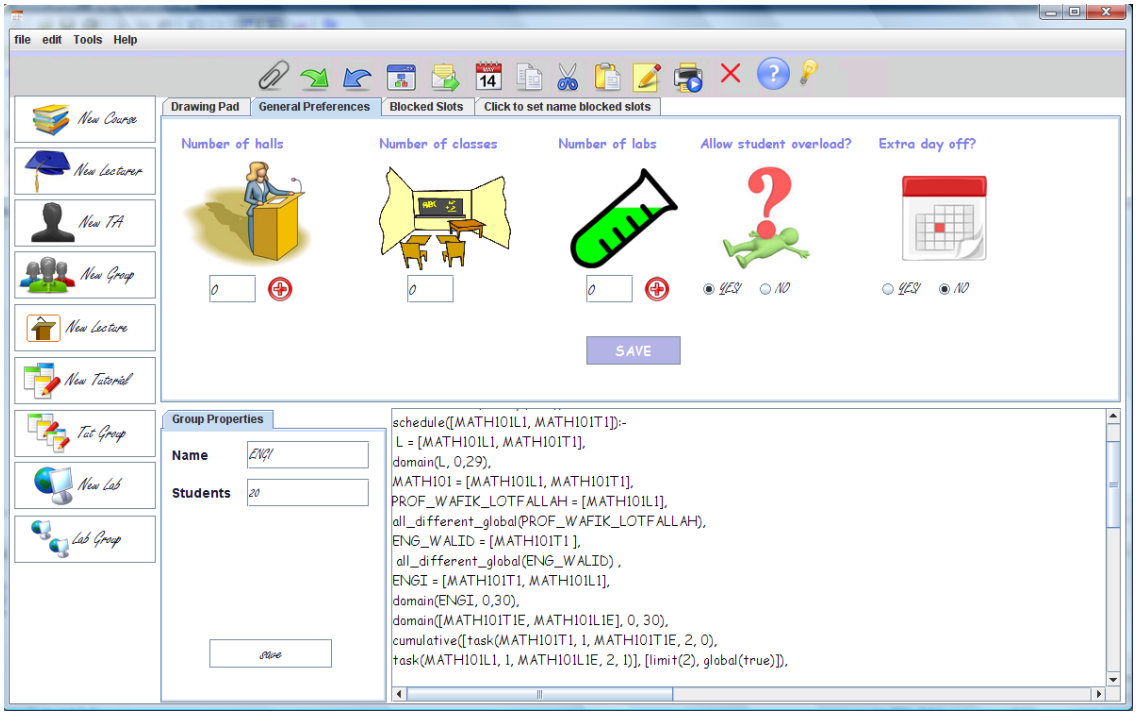


Fig. 5. The system with the preferences tab selected

would mean that on each day a study group can either have a class on the first time slot, or a class on the last time slot of that day but not both. Applying this to our problem would result on the following constraints being generated as the user creates the study group nodes. The `count` predicate is a SICStus constraint. If the first line below it can be read as: the list of events in `GROUP1` is constrained to have `GROUP1S0C` variables (Boolean) with value ”0” (meaning taking place in the first slot of Saturday). It also constrains the events to take place only in the first or last slot of the day (line 3):

```
%% Group1 will generate
count(0, GROUP1, GROUP1S0C),
count(2, GROUP1, GROUP1S2C),
GROUP1S0C + GROUP1S2C#=<1,
count(3, GROUP1, GROUP1S3C),
count(5, GROUP1, GROUP1S5C),
GROUP1S3C + GROUP1S5C#=<1,
count(6, GROUP1, GROUP1S6C),
count(8, GROUP1, GROUP1S8C),
GROUP1S6C + GROUP1S8C#=<1,

%% The same code will be generated for all groups
```

5.2 Dealing with time

The German University in Cairo requires lecturers to have Friday off and another day off (not necessarily Saturday). Thus, each lecturer should have one extra day where there is no scheduled events. Applying this to our problem would mean that either no lectures are given on the first day or no lectures are given on the second day or no lectures are given on the third day, etc. The end-user specifies the lecturer’s day off using the following tab in Fig. 6. It shows the time line and allows the user to block slots by simply clicking on the required slot.
If a lecturer wishes that no lectures should be given in the first slot on Saturday as illustrated in Fig. 6, then the following fragment of code will be generated:

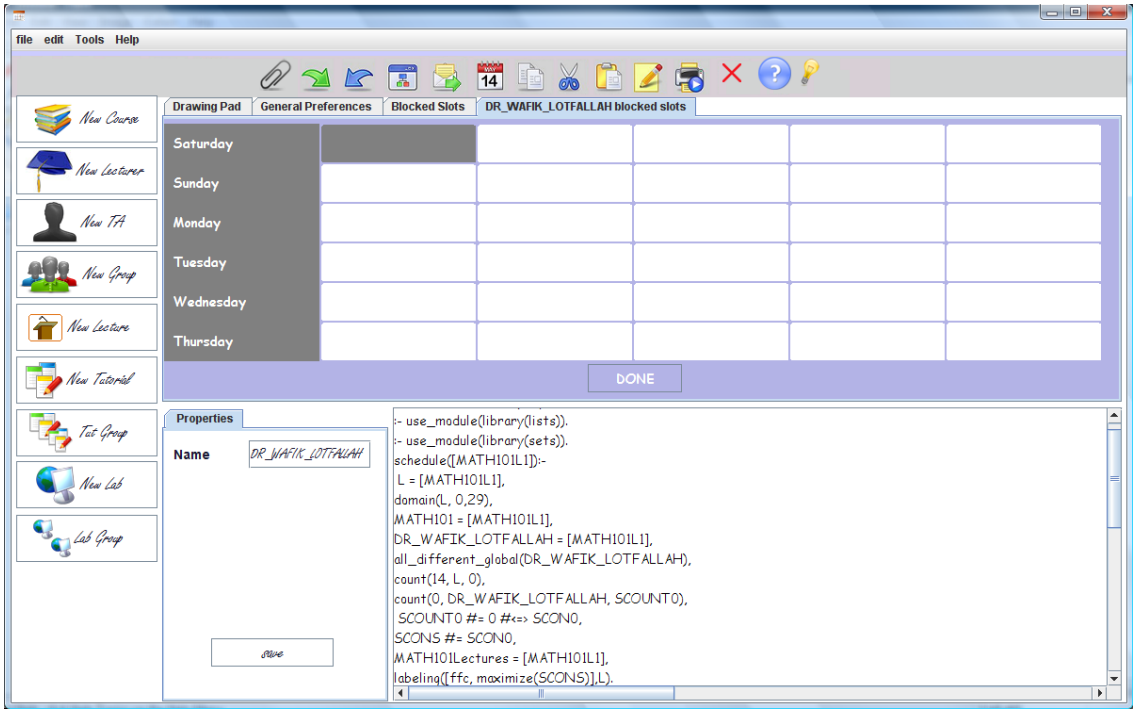


Fig. 6. The time tab of a lecturer with a single slot blocked

```
count(0, DR_WAFIK_LOTFALLAH, SCOUNT0),
SCOUNT0 #= 0 #<=> SCON0,
SCONS #= SCON0,
```

`SCOUNT0` is the number of 0s in the list of the lecturer `DR_WAFIK LOTFALLAH`. `SCOUNT0` being equal to 0 would result in the flag $SCON0$ being set to 1. `SCONS` is supposed to be the sum of the satisfied soft constraints. In this example, we only have one soft constraint thus $SCONS$ is equal to `SCON0`. As mentioned in Section 3, the aim of the timetabling is to find a solution that maximizes the number of lecturers' wishes. Thus, the labeling part will be modified as follows:

```
labeling([ffc, maximize(SCONS)],L).
```

Where `fcc` is the heuristic to select the variable with the smallest domain, breaking ties by selecting the variable that has the most constraints suspended on it and selecting the leftmost one.

6 System scalability and evaluation

In addition to the use of global constraints, a key asset of this approach is its modularity and scalability. With a focus on the resources and their direct links we can easily change properties of any created icon or add new links to the graph. This will lead to a dynamic update of the CP model. From the constructed graph, direct and indirect relations (e.g which lecturer teaches which study group) between events and resources can be clearly understood. Looking at the diagram one can clearly see all the components of a UTT instance. This is not the case if the problem is modelled as a constraint network [4]. Fig. 7 represents a constraint network with four variables. From the network, one can clearly understand that variables `COURSE1L1` and `COURSE2L1` can not be equal. However, one cannot understand the reason behind it, since on the network there is no difference between these two variables and any other two unequal variables in their representation. Either the problem description or the constructed problem model should be consulted to understand why such relation exists. Since the people involved in the university timetabling

process are usually not programming experts, a visual representation would require a clear representation of the problem through which the problem can be clearly understood by all stakeholders which is the case using our visual entity-relationship model.

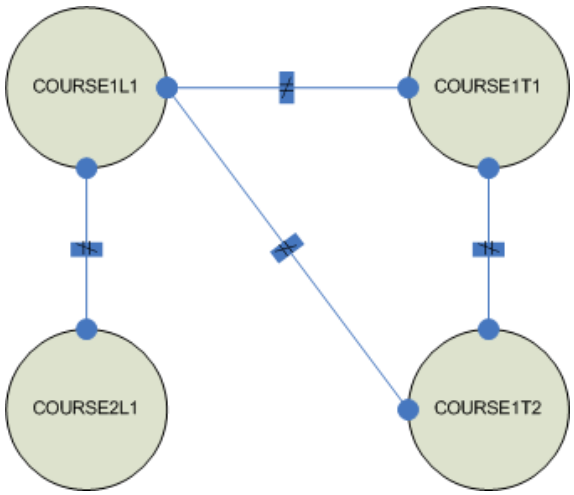


Fig. 7. A constraint network representing the timetabling problem

The system was used to generate the GUC university timetabling for one term. It was run on an Intel(R) Core(Tm) 2 Duo , with CPU 2.4 GHz with 2GB ram. The problem corresponds to 2233 events to be scheduled, over 400 resources, including 224 study groups and 201 lecturers and teaching assistants.
For this instance, the visual entity-relationship model was drawn in 5 hours. The corresponding SICStus code was generated in 7 seconds from the constructed graph. The first 5 solutions were found after three and half minutes. However, a manually generated schedule takes in general two to three months to be constructed by one timetable specialist.

7 Conclusion and Future Work

We have presented a visual graphical approach to specify combinatorial problems with application to university timetabling. The main contribution lies in the usage of type nodes to be distinguished from constraint graph variable nodes, and represent resources and events of the problem at hand. The rich semantic of the nodes and the links created among them can be used to generate efficient CP models using global constraints in a dynamic setting.
We intend to improve the representation of the graphs by providing additional features to group a number of icons and to combine them into one icon with the ability to expand and collapse that icon. Additionally, an interesting direction for future work is to investigate how the proposed approach can be generalized to handle any resource allocation problem by adding a generic interface to express application specific constraints.

References

1. S. Abdennadher, M. Aly, and M. Edward. Constraint-based timetabling system for the german university in cairo. In *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007*. Springer LNCS, 2007.
2. A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernandez, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints,*, 13(3), 2008.
3. T. Muller. ITC2007 solver description: a hybrid approach. *Annals of Operations Research*, 172(1), 2007.

4. M. Paltrinieri. A visual environment for constraint programming. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, 1995.
5. H. Rudov and L. Matyska. Constraint-based timetabling with student schedules, 2000.
6. E. Tsang. *Foundation of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

Warranted Derivations of Preferred Answer Sets

Ján Šefránek and Alexander Šimko

Comenius University, Bratislava, Slovakia; [sefranek,simko]@fmph.uniba.sk

Abstract. We are aiming at a semantics of logic programs with preferences defined on rules, which always selects a preferred answer set, if there is a non-empty set of (standard) answer sets of the given program.
It is shown in a seminal paper by Brewka and Eiter that the goal mentioned above is incompatible with their second principle and it is not satisfied in their semantics of prioritized logic programs. Similarly, also according to other established semantics, based on a prescriptive approach, there are programs with standard answer sets, but without preferred answer sets.
According to the standard prescriptive approach no rule can be fired before a more preferred rule, unless the more preferred rule is blocked. This is a rather imperative approach, in its spirit.
In our approach, rules can be blocked by more preferred rules, but the rules which are not blocked are handled in a more declarative style, their execution does not depend on the given preference relation on the rules.
An argumentation framework (different from the Dung's framework) is proposed in this paper. Argumentation structures are derived from the rules of a given program. An attack relation on argumentation structures is defined, which is derived from attacks of more preferred rules against the less preferred rules. Preferred answer sets correspond to complete argumentation structures, which are not blocked by other complete argumentation structures.

Keywords: extended logic program, answer set, preference, preferred answer set, argumentation structure

1 Introduction

The meaning of a nonmonotonic theory is often characterized by a set of *(alternative) belief sets*. It is appropriate to select sometimes some of the belief sets as more *preferred*.

We are focused in this paper on *extended logic programs* with a preference relation on rules, see, e.g., [1, 3, 9, 15]. Such programs are denoted by the term *prioritized* logic programs in this paper.

It is suitable to require that some preferred answer sets can be *selected* from a non-empty set of standard answer sets of a (prioritized) logic program.

Unfortunately, there are prioritized logic programs with standard answer sets, but without preferred answer sets according to the semantics of [1] and also of [3] or [15]. This feature is a consequence of the *prescriptive* approach to preference handling [4]. According to that approach no rule can be fired before a more preferred rule, unless the more preferred rule is blocked. This is a rather imperative approach, in its spirit.

An investigation of basic *principles* which should be satisfied by any system containing a preference relation on defeasible rules is of fundamental importance. This type of research has been initialized in the seminal paper [1]. Two basic principles are accepted in the paper.

The second of the principles is violated, if a function is assumed, which always selects a non-empty subset of preferred answer sets from a non-empty set of all standard answer sets of a prioritized logic program.

We believe that the possibility to select always a preferred answer set from a non-empty set of standard answer sets is of critical importance. This goal requires to accept a *descriptive* approach to preference handling. The approach is characterized in [5, 4] as follows: The order in which rules are applied, reflects their "desirability" – it is desirable to apply the most preferred rules.

In our approach, rules can be *blocked* by more preferred rules, but the rules which are not blocked are handled in a more declarative style. If we express this in terms of desirability, it is desirable to apply all

(applicable) rules, which are not blocked by a more preferred rule. The execution of non-blocked rules does not depend on their order. Dependencies of more preferred rules on less preferred rules do not prevent the execution of non-blocked rules.

Our goal is:

- to modify the Principles proposed by [1] in such a way that they do not contradict a selection of a non-empty set of preferred answer sets from the underlying non-empty set of standard answer sets,
- to introduce a notion of preferred answer sets that satisfies the above mentioned modification.

The proposed method is inspired by [7]. While there defeasible rules are treated as (defeasible) arguments, here (defeasible) assumptions, sets of default negations, are considered as arguments. Reasoning about preferences in a logic program is here understood as a kind of argumentation. Sets of default literals can be viewed as defeasible arguments, which may be contradicted by consequences of some applicable rules. The preference relation on rules is used in order to ignore the attacks of less preferred arguments against more preferred arguments. The core problem is to transfer the preference relation defined on rules to argumentation structures and, consequently, to answer sets.[1]

The basic argumentation structures correspond to the rules of a given program. Derivation rules, which enable derivation of argumentation structures from the basic ones are defined. That derivation leads from the basic argumentation structures (corresponding to the rules of a given program) to argumentation structures corresponding to the rules of an negative equivalent of the given program introduced in [6].

Attacks of more preferred rules against the less preferred rules are transferred via another set of derivation rules to the attacks between argumentation structures. Preferred answer sets are defined over that background. They correspond to complete and non-blocked (warranted) argumentation structures.

The contributions of this paper are summarized as follows. A modified set of principles for preferred answer set specification is proposed. A new type of argumentation framework is constructed, which enables a selection of preferred answer sets. There are basic arguments (argumentation structures) and attacks in the framework and also derived arguments and attacks. Rules for derivation of argumentation structures and rules for derivation of attacks of some argumentation structures against other argumentation structures are defined. Preferred answer sets are defined in terms of complete and non-blocked (warranted) argumentation structures. Finally, we emphasize that each program with a non-empty set of answer sets has a preferred answer set.

A preliminary version of the presented research has been published in [11]. The main differences between the preliminary and the current version are summarized in the Conclusions.[2] An extended version of this paper with proofs is accessible as [12].

2 Preliminaries

The language of extended logic programs is used in this paper.

Let At be a set of atoms. The set of *objective literals* is defined as $Obj = At \cup \{\neg\ A : A \in At\}$. If L is an objective literal then the expression of the form $not\ L$ is called *default* literal. Notation: $Def = \{not\ L \mid L \in Obj\}$. The set of literals Lit is defined as $Obj \cup Def$.

A convention: $\neg\neg A = A$, where $A \in At$. If X is a set of objective literals, then $not\ X = \{not\ L \mid L \in X\}$.

A *rule* is each expression of the form $L \leftarrow L_1, \ldots, L_k$, where $k \geq 0$, $L \in Obj$ and $L_i \in Lit$. If r is a rule of the form as above, then L is denoted by $head(r)$ and $\{L_1, \ldots, L_k\}$ by $body(r)$. If R is a set of rules, then $head(R) = \{head(r) \mid r \in R\}$ and $body(R) = \{body(r) \mid r \in R\}$. A finite set of rules is called *extended logic program* (program hereafter).

[1] Our intuitions connected to the notion of argumentation structure and also the used constructions are different from Dung's arguments or from arguments of [7, 2]. On the other hand, we plan an elaboration of presented constructions aiming at a theory, which generalizes Dung's abstract argumentation framework, TMS, constructions given, e.g., by [7] or [2]. Anyway, this paper does not present a contribution to argumentation theory.

[2] They are described in technical terms, assuming a familiarity with this paper. Most importantly, the notion of preferred answer set is changed in this paper.

The set of *conflicting literals* is defined as $CON = \{(L_1, L_2) \mid \ L_1 = not\ L_2 \vee L_1 = \neg L_2\}$. A set of literals S is *consistent* if $(S \times S) \cap CON = \emptyset$. An *interpretation* is a consistent set of literals. A *total* interpretation is an interpretation I such that for each objective literal L either $L \in I$ or $not\ L \in I$. A literal L is *satisfied* in an interpretation I iff $L \in I$ (notation: $I \models L$). A set of literals S is satisfied in I iff $S \subseteq I$ (notation: $I \models S$). A rule r is satisfied in I iff $I \models head(r)$ whenever $I \models body(r)$.

If S is a set of literals, then we denote $S \cap Obj$ by S^+ and $S \cap Def$ by S^-. Symbols $(body(r))^-$ and $(body(r))^+$ are used here in that sense (notice that the usual meaning of $body^-(r)$ is different). If $X \subseteq Def$ then $pos(X) = \{L \in Obj \mid not\ L \in X\}$. Hence, $not\ pos((body(r))^-) = (body(r))^-$. If r is a rule, then the rule $head(r) \leftarrow (body(r))^+$ is denoted by r^+.

An answer set of a program can be defined as follows (only consistent answer sets are defined).

A total interpretation S is an *answer set* of a program P iff S^+ is the least model[3] of the program $P^+ = \{r^+ \mid S \models (body(r))^-\}$. Note that an answer set S is usually represented by S^+ (this convention is sometimes used also in this paper).

The set of all answer sets of a program P is denoted by $AS(P)$. A program is called *coherent* iff it has an answer set.

Strict partial order is a binary relation, which is irreflexive, transitive and, consequently, asymmetric.

A *prioritized logic program* is usually defined as a triple $(P, \prec, \mathcal{N})$, where P is a program, $\prec$ a strict partial order on rules of P and a function $\mathcal{N}$ assigns names to rules of P. If $r_1 \prec r_2$ it is said that r_2 is more preferred than r_1.

We will not use $\mathcal{N}$. If a symbol r is used in this paper in order to denote a rule, then r is considered as the name of that rule (no different name $\mathcal{N}(r)$ is introduced).

3 Argumentation Structures

Our aim is to transfer a preference relation defined on rules to a preference relation on answer sets and, finally, to a notion of preferred answer sets. To that end argumentation structures are introduced. The basic argumentation structures correspond to rules. Some more general types of argumentation structures are derived from the basic argumentation structures. A special type of argumentation structures corresponds to answer sets.

Definition 1 ($\ll_P$, [10]) An objective literal L *depends* on a set of default literals $W \subseteq Def$ *with respect to a program* P ($L \ll_P W$[4]) iff there is a sequence of rules $\langle r_1, \ldots, r_k \rangle$, $k \geq 1$, $r_i \in P$ such that

- $head(r_k) = L$,
- $W \models body(r_1)$,
- for each i, $1 \leq i < k$, $W \cup \{head(r_1), \ldots, head(r_i)\} \models body(r_{i+1})$.

The set $\{L \in Lit \mid L \ll_P W\} \cup W$ is denoted by $Cn_{\ll_P}(W)$.
$W \subseteq Def$ is *self-consistent* w.r.t. a program P iff $Cn_{\ll_P}(W)$ is consistent. □

If $Z \subseteq Obj$, we will use sometimes the notation $Cn_{\ll_{P \cup Z}}(W)$, assuming that the program P is extended by the set of facts Z.

Definition 2 (Dependency structure) Let P be a program.

A self-consistent set $X \subseteq Def$ is called an *argument* w.r.t. the program P for a consistent set of objective literals Y, given a set of objective literals Z iff

1. $pos(X) \cap Z = \emptyset$,
2. $Y \subseteq Cn_{\ll_{P \cup Z}}(X)$.

We will use the notation $\langle Y \hookleftarrow X; Z \rangle$ and the triple denoted by it is called a *dependency structure* (w.r.t. P). □

[3] P^+ is treated as definite logic program, i.e., each objective literal of the form $\neg A$, where $A \in At$, is considered as a new atom.

[4] $L \ll_P W$ could be defined as $T_P^\omega(W)$ and $Cn_{\ll_P}(W)$ as $T_P^\omega(W)$

If $Z = \emptyset$ also a shortened notation $\langle Y \hookleftarrow X \rangle$ can be used. We will omit sometimes the phrase "w.r.t. P" and speak simply about dependency structures and arguments, if the corresponding program is clear from the context.

Basic argumentation structures comply with Definition 2 of dependency structures, if some conditions are satisfied:

Definition 3 (Basic argumentation structure) Let $r \in P$ be a rule such that

- $(body(r))^-$ is self-consistent and
- $pos((body(r))^-) \cap (body(r))^+ = \emptyset$.

Then $\mathcal{A} = \langle \{head(r)\} \hookleftarrow (body(r))^-; (body(r))^+ \rangle$ is called a *basic* argumentation structure. □

Proposition 4 *Each basic argumentation structure is a dependency structure.* □

We emphasize that only *self-consistent* arguments for *consistent* sets of objective literals are considered in this paper. Hence, programs as $P = \{p \leftarrow not\ p\}$ or $Q = \{p \leftarrow not\ q; \neg p \leftarrow not\ q\}$ are irrelevant for our constructions.

Some dependency structures can be derived from the basic argumentation structures. Only the dependency structures derived from the basic argumentation structures using derivation rules from Definition 5 are of interest in the rest of this paper, we will use the term *argumentation structure* for dependency structures derived from basic argumentation structures using derivation rules.

Derivation rules are motivated in Example 6.

Definition 5 (Derivation rules and argumentation structures) Each basic argumentation structure is an argumentation structure. Let P be a program.

R1 Let $r_1, r_2 \in P$, $\mathcal{A}_1 = \langle \{head(r_1)\} \hookleftarrow X_1; Z_1 \rangle$ and $\mathcal{A}_2 = \langle \{head(r_2)\} \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$ be argumentation structures, $head(r_2) \in Z_1$, $X_1 \cup (body(r_2))^- \cup Z_1 \cup (body(r_2))^+ \cup \{head(r_1)\}$ be consistent and $X_1 \cup (body(r_2))^-$ be self-consistent.
Then also $\mathcal{A}_3 = \langle head(r_1) \hookleftarrow X_1 \cup (body(r_2))^-; (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+ \rangle$ is an argumentation structure. We also write $\mathcal{A}_3 = u(\mathcal{A}_1, \mathcal{A}_2)$. We define u as a symmetric relation: $u(\mathcal{A}_1, \mathcal{A}_2) = u(\mathcal{A}_2, \mathcal{A}_1)$[5]

R2 Let $\mathcal{A}_1 = \langle Y_1 \hookleftarrow X_1 \rangle$ and $\mathcal{A}_2 = \langle Y_2 \hookleftarrow X_2 \rangle$ be argumentation structures and $X_1 \cup X_2 \cup Y_1 \cup Y_2$ be consistent and $X_1 \cup X_2$ be self-consistent.
Then also $\mathcal{A}_3 = \langle Y_1 \cup Y_2 \hookleftarrow X_1 \cup X_2 \rangle$ is an argumentation structure. We also write $\mathcal{A}_3 = \mathcal{A}_1 \cup \mathcal{A}_2$.

R3 Let $\mathcal{A}_1 = \langle Y_1 \hookleftarrow X_1 \rangle$ be an argumentation structure and $W \subseteq Def$.
If $X_1 \cup W \cup Y_1$ is consistent and $X_1 \cup W$ is self-consistent, then also $\mathcal{A}_2 = \langle Y_1 \hookleftarrow X_1 \cup W \rangle$ is an argumentation structure. We also write $\mathcal{A}_2 = \mathcal{A}_1 \cup W$.□

Example 6 ([1]) Let a program P be given (P is used as a running example in this paper):

r_1 $\quad b \leftarrow a, not\ \neg b$
r_2 $\quad \neg b \leftarrow not\ b$
r_3 $\quad a \leftarrow not\ \neg a.$

Suppose that $\prec = \{(r_2, r_1)\}$.

Consider the rule r_2. The default negation $not\ b$ plays the role of a *defeasible argument*. If the argument can be consistently evaluated as true with respect to a program containing r_2, then also $\neg b$ can (and must) be evaluated as true.

As regards the rule r_1, default negation $not\ \neg b$ can be treated as an argument for b, if a is true, it is an example of a "conditional argument".

The following basic argumentation structures correspond to the rules of P:
$\langle \{b\} \hookleftarrow \{not\ \neg b\}; \{a\} \rangle$,$\langle \{\neg b\} \hookleftarrow \{not\ b\} \rangle$, $\langle \{a\} \hookleftarrow \{not\ \neg a\} \rangle$. Let us denote them by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, respectively.

[5] Symmetry of u enables below a more succinct formulation of derivation rules Q1, Q2. The symbol u indicates that $\mathcal{A}_3$ is a result of an unfolding.

Some arguments can be treated as counterarguments against other arguments. If we accept the argument $not\ b$ (with the consequence $\neg b$), it can be treated as a counterargument to $not\ \neg b$ and, similarly, $not\ \neg b$ (with the consequence b, if a is true) as a counterargument against $not\ b$. On the level of argumentation structures it can be said that $\mathcal{A}_1$ contradicts $\mathcal{A}_2$ and vice versa.

The preference relation can be directly transferred to *basic* argumentation structures, hence $\mathcal{A}_1$ is more preferred than $\mathcal{A}_2$. Consequently, only the attack of $\mathcal{A}_1$ against $\mathcal{A}_2$ is relevant.

An example of a derived argumentation structure: $\mathcal{A}_3$ enables to "unfold" the condition a in $\mathcal{A}_1$, the resulting argumentation structure can be expressed as $\mathcal{A}_4 = \langle\{b\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle$. Similarly, $\mathcal{A}_5 = \langle\{a, b\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle$ can be derived from $\mathcal{A}_3$ and $\mathcal{A}_4$, $\mathcal{A}_5 = \mathcal{A}_3 \cup \mathcal{A}_4$.

We will also transfer the attack relation from the basic argumentation structures to the derived argumentation structures.

Observe that some argumentation structures correspond to answer sets. $\mathcal{A}_5$ corresponds to the answer set $\{a, b\}$ and $\mathcal{A}_6 = \langle\{a, \neg b\} \hookleftarrow \{not\ b, not\ \neg a\}\rangle$ to $\{a, \neg b\}$. Notice that $\mathcal{A}_6 = \mathcal{A}_2 \cup \mathcal{A}_3$. The attack relation enables to select the preferred answer set. This will be discussed later in Example 19. □

Proposition 7 *Each argumentation structure is a dependency structure.*

Proof. We have to show that an application of R1, R2 and R3 preserves properties of dependency structures.

R1 Since $S_1 = X_1 \cup (body(r_2))^- \cup Z_1 \cup (body(r_2))^+ \cup \{head(r_1)\}$ is consistent $S_2 = X_1 \cup (body(r_2))^- \cup (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+ \subseteq S_1$ is also consistent. It means $pos(X_1 \cup (body(r_2))^-) \cap ((Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+) = \emptyset$.
Let $Q = P \cup (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+$ and $w = head(r_2) \leftarrow$.
From $head(r_2) \in Cn_{\ll_{P \cup (body(r_2))^+}}((body(r_2))^-)$ we have sequence of rules $R_2 = \langle q_1, q_2, \ldots, q_m\rangle$ where $m > 0$ and $q_m = r_2$.
From $head(r_1) \in Cn_{\ll_{P \cup Z_1}}(X_1)$ we have sequence of rules $R_1 = \langle p_1, p_2, \ldots, p_n\rangle$ where $n > 0$ and $p_n = r_1$. We assume there is at most one occurrence of w in R_1. Otherwise we can remove all but leftmost one. Note that since $r_2 \in P$ there is a possibility to satisfy $body(r_1)$ in a different way than using w.
If $w \in R_1$ then we have $p_i = w$ for some $1 \leq i < n$. We construct sequence
$R_3 = \langle q_1, q_2, \ldots, q_m, p_1, p_2, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\rangle$. If $w \notin R_1$ we construct sequence
$R_3 = \langle q_1, q_2, \ldots, q_m, p_1, p_2, \ldots, p_n\rangle$. In both cases R_3 satisfy conditions from definition 1 for assumption $X_1 \cup (body(r_2))^-$.
Since rules in R3 are from program Q we have $head(r_1) \in Cn_{\ll_Q}(X_1 \cup (body(r_2))^-)$.

R3 $Z_2 = \emptyset$ hence $pos(X_1 \cup W) \cap Z_2 = \emptyset$. We have $Y_1 \subseteq Cn_{\ll_P}(X_1)$. So for every $y \in Y_1$ there is a sequence R of rules that satisfy Definition 1 for assumption X_1. Same sequence satisfy definition 1 for superset assumption $X_1 \cup W$. Hence $y \in Cn_{\ll_P}(X_1 \cup W)$ and $Y_1 \subseteq Cn_{\ll_P}(X_1 \cup W)$.

R2 $Z_3 = \emptyset$ hence $pos(X_1 \cup X_2) \cap Z_3 = \emptyset$. We have $Y_1 \subseteq Cn_{\ll_P}(X_1)$ hence $Y_1 \subseteq Cn_{\ll_P}(X_1 \cup X_2)$. We also have $Y_2 \subseteq Cn_{\ll_P}(X_2)$ hence $Y_2 \subseteq Cn_{\ll_P}(X_1 \cup X_2)$. Therefore $Y_1 \cup Y_2 \subseteq Cn_{\ll_P}(X_1 \cup X_2)$.

A *derivation* of an argumentation structure $\mathcal{A}$ (w.r.t. P) is a sequence $\langle\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\rangle$ of argumentation structures (w.r.t. P) such that $\mathcal{A}_1$ is a basic argumentation structure, $\mathcal{A} = \mathcal{A}_k$ and each $\mathcal{A}_i$, $1 < i \leq k$, is either a basic argumentation structure or it is obtained by R1 or R2 or R3 from preceding argumentation structures.

4 Attacks

Our approach to preferred answer sets is based on a solution of conflicts between argumentation structures. We distinguish three steps towards that goal. *Contradictions* between argumentation structures represent the elementary step. Rule preference and contradiction between basic argumentation structures are used to form an attack relation. Consider two basic argumentation structures $\mathcal{A}_1$ and $\mathcal{A}_2$. If $\mathcal{A}_1$ contradicts $\mathcal{A}_2$ and corresponds to a more preferred rule, then it *attacks* $\mathcal{A}_2$. Attacks are propagated to other argumentation structures using derivation rules. Attacks between argumentation structures depend on how argumentation structures are derived. Hence, we need a more context-independent notion and we define a relation of *blocking* between argumentation structures. The complement of blocking (warranting) is used in the definition of preferred argumentation structures.

Definition 8 Consider argumentation structures $\mathcal{A} = \langle Y_1 \hookleftarrow X_1; Z_1 \rangle$ and $\mathcal{B} = \langle Y_2 \hookleftarrow X_2; Z_2 \rangle$.

If there is a literal $L \in Y_1$ such that $not\ L \in X_2$, it is said that the argument X_1 *contradicts* the argument X_2 and the argumentation structure $\mathcal{A}$ *contradicts* the argumentation structure $\mathcal{B}$.

It is said that X_1 is a *counterargument* to X_2. □

The basic argumentation structures corresponding to the facts of the given program are not contradicted.

Let $r_1 = a \leftarrow$ be a fact and $not\ a \in (body(r_2))^-$. Then any $W \subseteq Def$ s.t. $(body(r_2))^- \subseteq W$ is not self-consistent and, therefore, it is not an argument.

Example 9 In Example 6, $\mathcal{A}_1$ contradicts $\mathcal{A}_2$ and $\mathcal{A}_2$ contradicts $\mathcal{A}_1$.

Only some counterarguments are interesting: the rule r_1 is more preferred than the rule r_2, therefore the counterargument of $\mathcal{A}_2$ against $\mathcal{A}_1$ should not be "effectual". We are going to introduce a notion of *attack* in order to denote "effectual" counterarguments. □

Similarly as for the case of argumentation structures, the basic attacks are defined first. A terminological convention: if $\mathcal{A}_1$ attacks $\mathcal{A}_2$, it is said that the pair $(\mathcal{A}_1, \mathcal{A}_2)$ is an attack.

Definition 10 Let $r_2 \prec r_1$ and let $\mathcal{A}_1 = \langle \{head(r_1)\} \hookleftarrow (body(r_1))^-; (body(r_1))^+ \rangle$ contradicts $\mathcal{A}_2 = \langle \{head(r_2)\} \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$.

Then $\mathcal{A}_1$ *attacks* $\mathcal{A}_2$ and it is said that this attack is *basic*. □

Attacks between argumentation structures "inherited" (propagated) from basic attacks are defined in terms of derivation rules. The rules of that inheritance are motivated and defined below.

Example 11 Let us continue with Example 6.

Consider the basic argumentation structures $\mathcal{A}_1 = \langle \{b\} \hookleftarrow \{not\ \neg b\}; \{a\} \rangle$, $\mathcal{A}_2 = \langle \{\neg b\} \hookleftarrow \{not\ b\} \rangle$, $\mathcal{A}_3 = \langle \{a\} \hookleftarrow \{not\ \neg a\} \rangle$ and the derived argumentation structures $\mathcal{A}_4 = \langle \{b\} \hookleftarrow \{not\ \neg b, not\ \neg a\} \rangle$, $\mathcal{A}_5 = \langle \{b, a\} \hookleftarrow \{not\ \neg b, not\ \neg a\} \rangle$, $\mathcal{A}_6 = \langle \{\neg b, a\} \hookleftarrow \{not\ b, not\ \neg a\} \rangle$.

$(\mathcal{A}_1, \mathcal{A}_2)$ is the only basic attack (the more preferred $\mathcal{A}_1$ attacks the less preferred $\mathcal{A}_2$).

Derivations of the attacks of $(\mathcal{A}_4, \mathcal{A}_2)$ and $(\mathcal{A}_5, \mathcal{A}_2)$ could be motivated as follows. $\mathcal{A}_4$ is derived from $\mathcal{A}_1$ and $\mathcal{A}_3$ using R1, the attack of $\mathcal{A}_1$ against $\mathcal{A}_2$ should be propagated to the attack $(\mathcal{A}_4, \mathcal{A}_2)$. Note that $\mathcal{A}_3$ is not attacked.

Now, $\mathcal{A}_5$ is derived from $\mathcal{A}_3$ and $\mathcal{A}_4$. Again, the attack of $\mathcal{A}_4$ against $\mathcal{A}_2$ should be inherited by $(\mathcal{A}_5, \mathcal{A}_2)$.

Similarly, $\mathcal{A}_6$ is derived from attacked $\mathcal{A}_2$. The attacks against $\mathcal{A}_2$ are transferred to the attacks against $\mathcal{A}_6$. The attack $(\mathcal{A}_5, \mathcal{A}_6)$ is a crucial one, a selection of preferred answer set is based on it; compare with Example 19.

On the contrary, $\mathcal{A}_2$ contradicts $\mathcal{A}_4$ and $\mathcal{A}_5$, but it is based on a less preferred rule, hence those contradictions are not considered as attacks. □

First we define two rules, Q1 and Q2, which specify inheritance of attacks "via unfolding" - use of the rule R1. Second, two rules Q3 and Q4 derive attacks when the attacking or attacked side is joined with another argumentation structure - use of the rule R2. Finally, rules Q5 and Q6 derive attacks, if attacking or attacked side is extended by an assumption - use of the rule R3. Some asymmetries between pairs Q1, Q2 and Q3, Q4 will be discussed below, see Example 22.

Definition 12 (Attack derivation rules) Basic attacks are attacks.

Q1 Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be argumentation structures such that:
- $\mathcal{A}_1$ attacks $\mathcal{A}_2$,
- $\mathcal{A}_3$ does not attack $\mathcal{A}_1$, and
- $u(\mathcal{A}_2, \mathcal{A}_3)$ is argumentation structure.

Then $\mathcal{A}_1$ attacks $u(\mathcal{A}_2, \mathcal{A}_3)$.

Q2 Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be argumentation structures such that:
- $\mathcal{A}_1$ attacks $\mathcal{A}_2$,
- $\mathcal{A}_3$ is not attacked, and

– $u(\mathcal{A}_1, \mathcal{A}_3)$ is argumentation structure.

Then $u(\mathcal{A}_1, \mathcal{A}_3)$ attacks $\mathcal{A}_2$.

Q3 Let $\mathcal{A}_1$ and $\mathcal{A}_3$ be argumentation structures of the form $\langle X \hookleftarrow Y \rangle$ and $\mathcal{A}_2$ be an argumentation structure. Suppose that:
– $\mathcal{A}_1$ attacks $\mathcal{A}_2$,
– $\mathcal{A}_3$ is not attacked and
– $\mathcal{A}_1 \cup \mathcal{A}_3$ is argumentation structure.

Then $\mathcal{A}_1 \cup \mathcal{A}_3$ attacks $\mathcal{A}_2$.

Q4 Let $\mathcal{A}_1$ be an argumentation structure and $\mathcal{A}_2$, $\mathcal{A}_3$ be argumentation structures of the form $\langle X \hookleftarrow Y \rangle$ such that:
– $\mathcal{A}_1$ attacks $\mathcal{A}_2$,
– $\mathcal{A}_3$ does not attack $\mathcal{A}_1$, and
– $\mathcal{A}_2 \cup \mathcal{A}_3$ is argumentation structure.

Then $\mathcal{A}_1$ attacks $\mathcal{A}_2 \cup \mathcal{A}_3$.

Q5 Let $\mathcal{A}_1 = \langle X \hookleftarrow Y \rangle$ and $\mathcal{A}_2$ be argumentation structures. Let $W \subseteq Def$. Suppose that:
– $\mathcal{A}_1$ attacks $\mathcal{A}_2$, and
– $\mathcal{A}_1 \cup W = \langle X \hookleftarrow Y \cup W \rangle$ is argumentation structure.

Then $\mathcal{A}_1 \cup W$ attacks $\mathcal{A}_2$.

Q6 Let $\mathcal{A}_1$ and $\mathcal{A}_2 = \langle X \hookleftarrow Y \rangle$ be argumentation structures. Let $W \subseteq Def$. Suppose that:
– $\mathcal{A}_1$ attacks $\mathcal{A}_2$, and
– $\langle X \hookleftarrow Y \cup W \rangle = \mathcal{A}_2 \cup W$ is argumentation structure.

Then $\mathcal{A}_1$ attacks $\mathcal{A}_2 \cup W$.

There are no other attacks except those specified above. □

Definition 13 A *derivation of an attack* $\mathcal{X}$ is a sequence $\mathcal{X}_1, \ldots, \mathcal{X}_k$, where $\mathcal{X} = \mathcal{X}_k$, each $\mathcal{X}_i$ is an attack (a pair of attacking and attacked argumentation structures), $\mathcal{X}_1$ is a basic attack and each $\mathcal{X}_i$, $1 < i \leq k$ is either a basic attack or it is derived from the previous attacks using rules Q1, Q2, Q3, Q4, Q5, Q6.

Derivations of argumentation structures and of attacks blend together. Example 15 shows that a pair of argumentation structures $(\mathcal{B}, \mathcal{A})$ is an attack w.r.t. a derivation, but it is not an attack w.r.t another derivation. Let us start with a definition.

Definition 14 Let a program P and an answer set S be given. Let be $R = \{r \in P \mid body(r) \subseteq S\}$. It is said that R is the set of all *generating rules* of S^+. □

Example 15 Let P be

r_1 $\quad a \leftarrow not\ b$
r_2 $\quad b \leftarrow not\ a$
r_3 $\quad a \leftarrow not\ c$
r_4 $\quad c \leftarrow b.$

$\prec = \{(r_1, r_2)\}$.

There are two answer sets of P: $S_1 = \{a\}$ and $S_2 = \{b, c\}$. The corresponding argumentation structures are $\mathcal{A} = \langle \{a\} \hookleftarrow \{not\ b, not\ c\} \rangle$ and $\mathcal{B} = \langle \{b, c\} \hookleftarrow \{not\ a\} \rangle$, respectively.

There are two derivations of $\mathcal{A}$. Both derivations start from a basic argumentation structure and R3 is used. The first is the sequence $\mathcal{A}_1, \mathcal{A}$ and the second is $\mathcal{A}_3, \mathcal{A}$, where $\mathcal{A}_1 = \langle \{a\} \hookleftarrow \{not\ b\} \rangle$ and $\mathcal{A}_3 = \langle \{a\} \hookleftarrow \{not\ c\} \rangle$.

If the sequence $\mathcal{A}_1, \mathcal{A}$ is considered, an attack against $\mathcal{A}$ is derivable. Let be $\mathcal{A}_2 = \langle \{b\} \hookleftarrow \{not\ a\} \rangle$, $\mathcal{A}_4 = \langle \{c\} \hookleftarrow \emptyset; \{b\} \} \rangle$. The corresponding attack derivation is as follows:
$(\mathcal{A}_2, \mathcal{A}_1), (u(\mathcal{A}_4, \mathcal{A}_2), \mathcal{A}_1), (\mathcal{B}, \mathcal{A}_1), (\mathcal{B}, \mathcal{A})$, where Q2, Q3 and Q6 are used.

The only basic attack of our example is $(\mathcal{A}_2, \mathcal{A}_1)$. Hence, the second derivation $\mathcal{A}_3, \mathcal{A}$ of $\mathcal{A}$ cannot be attacked.

The derivations of $\mathcal{A}$ correspond to two sets of rules generating S_1, i.e., $R_1 = \{r_1\}$, and $R_2 = \{r_3\}$. R_1 contains an attacked rule, while R_2 does not contain such a rule. We accept that if there is a set of rules generating an answer set S s.t. no rule is attacked by a rule generating another answer set, then we can consider S as a preferred one.

We transfer this rather credulous approach to derivations of preferred argumentation structures. □

Definition 16 (Complete arguments) An argumentation structure $\langle Y \hookleftarrow X \rangle$ is called *complete* iff for each literal $L \in Obj$ it holds that $L \in Y$ or $not\ L \in X$. □

Definition 17 (Warranted and blocked derivations) Let $\sigma = \mathcal{A}_1, \ldots, \mathcal{A}_k$ be a derivation of an argumentation structure $\mathcal{A}$, where $\mathcal{A} = \mathcal{A}_k$.

It is said that σ is *blocked* iff there is a derivation τ of the attack $(\mathcal{B}, \mathcal{A})$, where $\mathcal{B}$ is a complete argumentation structure and each member of τ contains an $\mathcal{A}_i$ as a second component.

A derivation is *warranted* if it is not blocked. □

Definition 18 (Warranted and blocked argumentation structures) An argumentation structure $\mathcal{A}$ is warranted iff there is a warranted derivation of $\mathcal{A}$.

$\mathcal{A}$ is blocked iff each derivation of $\mathcal{A}$ is blocked. □

5 Preferred answer sets

Example 19 Consider our running example, where we have complete argumentation structures $\mathcal{A}_5 = \langle \{b, a\} \hookleftarrow \{not\ \neg b, not\ \neg a\} \rangle, \mathcal{A}_6 = \langle \{\neg b, a\} \hookleftarrow \{not\ \neg a, not\ b\} \rangle$.

We will prefer $\mathcal{A}_5$ over $\mathcal{A}_6$. $\mathcal{A}_6$ is blocked by $\mathcal{A}_5$. On the other hand, $\mathcal{A}_5$ is not blocked.

Consequently, we will consider $\{a, b\}$ as a preferred answer set of the given prioritized logic program. □

Definition 20 (Preferred answer set) An argumentation structure is *preferred* iff it is complete and warranted.

$Y \cup X$ is a *preferred answer set* iff $\langle Y \hookleftarrow X \rangle$ is a preferred argumentation structure. □

Notice that our notion of preferred answer set is rather a credulous one, it is based on the notion of warranted derivation, i.e., at least one derivation of a preferred answer set should not be blocked.

The following example shows that the argumentation structure corresponding to the only answer set of a program is preferred, even if it is attacked (by an argumentation structure which is not complete).

Example 21

r_1	$b \leftarrow not\ a$
r_2	$a \leftarrow not\ b$
r_3	$c \leftarrow a$
r_4	$c \leftarrow not\ c$

$\prec = \{(r_1, r_2), (r_3, r_4)\}$.

Let the basic argumentation structures be denoted by $\mathcal{A}_i$, $i = 1, \ldots, 4$. $(\mathcal{A}_1, \mathcal{A}_2), (\mathcal{A}_3, \mathcal{A}_4)$ are the basic attacks. $\mathcal{A}_1$ attacks $\mathcal{A}_5 = \langle \{c\} \hookleftarrow \{not\ b\} \rangle$ according to the rule Q1 and $\mathcal{A}_1$ attacks $\mathcal{A}_6 = \langle \{c, a\} \hookleftarrow \{not\ b\} \rangle$ according to the rule Q4.

Remind that according to Definition 17 a derivation can be blocked only by a complete argumentation structure and an argumentation structure is blocked iff each its derivation is blocked. Consequently, the complete argumentation structure $\mathcal{A}_6$ is not blocked by another complete argumentation structure (there is no such structure) and, consequently, it is the preferred argumentation structure.

We distinguish between attacking and blocking. A blocked argumentation structure is attacked by a *complete* argumentation structure. Preferred argumentation structures are not blocked. □

Next example explains asymmetries between Q1, Q2 and Q3, Q4. The main idea is as follows. We are more cautious when an attacking argumentation structure is derived (Q2, Q3) and we require that a "parent" of the attacking argumentation structure is not attacked at all. On the other hand, a scheme of derivation rules Q1 and Q4 is as follows: $\mathcal{A}_1$ attacks $\mathcal{A}_2$, $\mathcal{A}$ is a derived argumentation structure from the attacked $\mathcal{A}_2$ and an argumentation structure $\mathcal{A}_3$. In order to derive an attack of $\mathcal{A}_1$ against $\mathcal{A}$ it is sufficient to assume that $\mathcal{A}_3$ does not attack $\mathcal{A}_1$. However, there are some problems with this design decision.

Example 22 Consider a program P:

r_1 $a_1 \leftarrow not\ a_3, not\ d_2$
r_2 $d_1 \leftarrow not\ a_3, not\ d_2$
r_3 $a_2 \leftarrow not\ a_1, not\ d_3$
r_4 $d_2 \leftarrow not\ a_1, not\ d_3$
r_5 $a_3 \leftarrow not\ a_2, not\ d_1$
r_6 $d_3 \leftarrow not\ a_2, not\ d_1$

$\prec = \{(r1, r4), (r3, r5), (r6, r2)\}$.

Notice that a complete argumentation structure $\mathcal{B}_1 = \langle\{a_1, d_1\} \hookleftarrow \{not\ a_3, not\ d_2\}\rangle$ is derived from $\mathcal{A}_1$ corresponding to r_1 and from $\mathcal{A}_2$ corresponding to r_2, similarly $\mathcal{B}_2 = \langle\{a_2, d_2\} \hookleftarrow \{not\ a_1, not\ d_3\}\rangle$ is derived from $\mathcal{A}_3$ corresponding to r_3 and from $\mathcal{A}_4$ corresponding to r_4 and $\mathcal{B}_3 = \langle\{a_3, d_3\} \hookleftarrow \{not\ a_2, not\ d_1\}\rangle$ is derived from $\mathcal{A}_5$ corresponding to r_5 and from $\mathcal{A}_6$ corresponding to r_6. $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ are all complete argumentation structures of our example.

Suppose that Q3 does not contain condition that a "parent" of the attacking argumentation structure is not attacked at all. Then we get that each complete argumentation structure is blocked, consequently, no preferred answer set can be selected. But we are extremely interested in a selection of a preferred answer set.

As a consequence, we are too liberal in selecting preferred answer sets: Consider program

r_1 $a \leftarrow$
r_2 $b \leftarrow not\ a$
r_3 $c \leftarrow not\ b$
r_4 $b \leftarrow not\ c$

$\prec = \{(r2, r3), (r3, r4)\}$.

We get that both $S_1 = \{a, c\}$ and $S_2 = \{a, b\}$ are preferred answer sets, but S_2 is not an intuitive selection. The reason is that both argumentation structures (let us denote them by $\mathcal{A}_1$ and $\mathcal{A}_2$) corresponding to S_1 and S_2, respectively, are attacked and rule Q3 cannot be applied. Hence, each derivation of $\mathcal{A}_1$ and $\mathcal{A}_2$ is warranted. A weaker version of Q3 would enable to repair this, however, it is a too high price for us. □

Theorem 23 *If S is a preferred answer set of $(P, \prec, \mathcal{N})$, then S is an answer set of P.*

6 Principles

The principles (partially) specify what it means that an order on answer sets corresponds to the given order on rules. The first two principles below are dependent on [1]. Principle III reproduces an idea of Proposition 6.1 from [1]. The Principles of [1] are originally expressed in an abstract way for the general case of nonmonotonic prioritized defeasible rules. We restrict the discussion (and the wording) of the Principles to the case of logic programs and answer sets.

Let P be a program and $r_1, r_2 \in P$. It is said that r_2 is *attacked* by r_1 (r_1 attacks r_2) iff $not\ head(r_1) \in (body(r_2))^-$.

Definition 24 Let a prioritized logic program $(P, \prec, \mathcal{N})$ be given. Let R_1, R_2 be sets of generating rules for some answer sets of P. It is said that R_1 *attacks* R_2 iff there is $r_1 \in R_1, r_2 \in R_2$ such that $r_2 \prec r_1$ and r_1 attacks r_2.

Definition 25 Let a prioritized logic program $(P, \prec, \mathcal{N})$ be given. Let R be a set of generating rules of some answer set of P. It is said that R is a *warranted set of generating rules* iff there is no set Q of generating rules of some answer set of P such that Q attacks R . □

Principle I in its original formulation does not hold for our approach. A terminological remark – words associated to our approach (attack, warranted) are used in presented formulations of Principles. But remind definitions 24 and 25 – the notions are defined for generating sets of rules independently on our approach. We have considered the same principles also in another approach, see [14] and also [13]. It is defined directly on generating sets, and uses neither argumentation structures nor derivation rules.

In all principles below it is assumed that a prioritized logic program $(P, \prec, \mathcal{N})$ is given.[6]

Principle I Let A_1 and A_2 be two answer sets of the program P. Let $R \subset P$ be a set of rules and $d_1, d_2 \in P \setminus R$ are rules. Let A_1^+, A_2^+ be generated by the rules $R \cup \{d_1\}$ and $R \cup \{d_2\}$, respectively. If d_1 is preferred over d_2 and each set of generating rules of $\mathcal{A}_2^+$ is attacked by a warranted set of generating rules of some answer set of P, then A_2 is not a preferred answer set of $(P, \prec, \mathcal{N})$. □

Our formulation of Principle I differs from the original formulation in [1] – the condition "each set of generating rules of $\mathcal{A}_2^+$ is attacked by a warranted set of generating rules of some answer set of P" is added because of the credulous stance to warranted derivations accepted in this paper

We do not accept the following principle. See discussion below.

Principle II Let A be a preferred answer set of a prioritized logic program $(P, \prec, \mathcal{N})$ and r be a rule such that $(body(r))^+ \not\subseteq A^+$. Then A is a preferred answer set of $(P \cup \{r\}, \prec', \mathcal{N}')$, whenever $\prec'$ agrees with $\prec$ on rules in P and $\mathcal{N}'$ extends $\mathcal{N}$ with the name r. □

We believe that the possibility to always select a preferred answer set from a non-empty set of standard answer sets is of critical importance.

Principle III Let $\mathcal{B} \neq \emptyset$ be the set of all answer sets of P. Then there is a selection function Σ s.t. $\Sigma(\mathcal{B})$ is the set of all preferred answer sets of $(P, \prec, \mathcal{N})$, where $\emptyset \neq \Sigma(\mathcal{B}) \subseteq \mathcal{B}$. □

It is shown in [1], Proposition 6.1, that Principle II is incompatible with Principle III, if the notion of preferred answer set from [1] is accepted:

Example 26 ([1]) Consider program P, whose single standard answer set is $S = \{b\}$ and the rule (1) is preferred over the rule (2).

$$c \leftarrow not\ b \tag{1}$$

$$b \leftarrow not\ a \tag{2}$$

S is not a preferred answer set in the framework of [1]. Assume that S, the only standard answer set of P, is selected – according to the Principle III – as the preferred answer set of $(P, \prec, \mathcal{N})$.[7] Let P' be $P \cup \{a \leftarrow c\}$ and $a \leftarrow c$ be preferred over the both rules 1 and 2. P' has two standard answer sets, S and $T = \{a, c\}$. Note that $\{c\} \not\subseteq S^+$. Hence, S should be the preferred answer set of P' according to the Principle II. However, in the framework of [1] the only preferred answer set of $(P', \prec', \mathcal{N}')$ is T. This selection of preferred answer set satisfies clear intuitions – T is generated by the two most preferred rules.

In our approach the complete argumentation structure $\mathcal{A}_4 = \langle \{a, c\} \hookleftarrow \{not\ b\} \rangle$ blocks the complete argumentation structure $\mathcal{A}_5 = \langle \{b\} \hookleftarrow \{not\ a, not\ c\} \rangle$, hence, $\mathcal{A}_4$ is preferred and $\{a, c\}$ is the preferred answer set.

Principle III is of crucial value according to our view. A more detailed justification of our decision not to accept Principle II is presented in [11]. We mention here only that we select preferred answer sets of P' from a broader variety of possibilities. Consequently, no condition satisfied by a subset of those possibilities should constrain the selection of preferred answer set from the extended set of possibilities. □

Principle II is not accepted also in [8]. According to [4] descriptive approaches do not satisfy this principle in general.

Principle IV expresses when an answer set is a preferred one. We consider it as an expression of a descriptive approach to preferred answer set specification, as we understand it and accept in this paper.

Principle IV Let S be answer set of P. Suppose that S is generated by a set of rules R. If R is a warranted set of generating rules then S is a preferred answer set. □

[6] The original formulation of principles by [1] is as follows.

Principle I. Let B_1 and B_2 be two belief sets of a prioritized theory $(T; \prec)$ generated by the (ground) rules $R \cup d_1$ and $R \cup d_2$, where $d_1, d_2 \notin R$, respectively. If d_1 is preferred over d_2, then B_2 is not a (maximally) preferred belief set of T.

Principle II. Let B be a preferred belief set of a prioritized theory $(T; \prec)$ and r a (ground) rule such that at least one prerequisite of r is not in B. Then B is a preferred belief set of $(T \cup \{r\}; \prec')$ whenever $\prec'$ agrees with $\prec$ on priorities among rules in T.

[7] Observe that the only derived complete argumentation structure is $\langle \{b\} \hookleftarrow \{not\ a, not\ c\} \rangle$. Hence, $\{b\}$ is a preferred answer set of $(P, \prec, \mathcal{N})$ in our framework.

As regards a choice of principles, we accept the position of [1]: even if somebody does not accept a set of principles for preferential reasoning, those (and similar) principles are still of interest as they may be used for classifying different patterns of reasoning.

7 Discussion

The question whether derivation rules for attacks are sufficient and necessary arises in a natural way. Our only response to the question in this paper is that Principles I, III, IV are satisfied, when we use notions of attack, blocking and warranting introduced in this paper We proceed to theorems about satisfaction of principles.

Theorem 27 *Principle III is satisfied. Let $\mathcal{P} = (P, \prec, \mathcal{N})$ be a prioritized logic program and $AS(P) \neq \emptyset$. Then there is a preferred answer set of $\mathcal{P}$.*

Proof. **Case 1** is trivial – if a program P have only one answer set S, then no complete argumentation structure blocks $\langle S^+ \hookleftarrow S^- \rangle$.

Case 2. Let a program P has only two answer sets S_1 and S_2. Let the corresponding complete argumentation structures be $\mathcal{A}_1 = \langle S_1^+ \hookleftarrow S_1^- \rangle$ and $\mathcal{A}_2 = \langle S_2^+ \hookleftarrow S_2^- \rangle$, respectively. Suppose that $\mathcal{A}_1$ and $\mathcal{A}_2$ block each other.

It means that each derivation of both is blocked by the other complete argumentation structure. Consider all derivations of $\mathcal{A}_1$ (which should be blocked by $\mathcal{A}_2$). Hence, each derivation σ_i contains an argumentation structure $\mathcal{B}_i$ attacked by $\mathcal{A}_2$, i.e., $\mathcal{X} = (\mathcal{A}_2, \mathcal{B}_i)$ is an attack. Each derivation of $\mathcal{X}$ should start from a basic attack and ends with $(\mathcal{A}_2, \mathcal{B}_i)$.

If $\mathcal{X}$ is a basic attack, then the only generating set of rules of S_2 contains only one rule $r = S_2^+ \leftarrow S_2^-$, where $S^+ = \{head(r)\}$. We assume that there is a rule r_1 s.t. $r_1 \prec r$ and $not\ head(r) \in (body(r_1))^-$. On the other hand, $\mathcal{A}_1$ blocks $\mathcal{A}_2$ and there is an $r_2 \in P$ which is among the generating rules of S_1, $r \prec r_2$ and $not\ head(r_2) \in (body(r))^-$.

Notice that $\langle head(r_2) \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$ attacks $\mathcal{A}_2$. If $(body(r_2))^+ \neq \emptyset$, then a derivation of attack $(\mathcal{A}_2, \mathcal{A}_1)$ has to use Q1 and $\langle head(r_2) \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$. But Q1 is not applicable – attacking argumentation structure should be not attacked. Similarly, if $(body(r_2))^+ = \emptyset$, Q4 should be used, but Q4 is not applicable because of the same reason.

Assume that $\mathcal{X}$ is not a basic attack. Then there is a basic attack as follows.
Let be $\mathcal{R}_1 = \langle head(r_1) \hookleftarrow (body(r_1))^-; (body(r_1))^+ \rangle$,
$\mathcal{R}_2 = \langle head(r_2) \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$, where
$head(r_2) \in S_2, head(r_1) \in S_1, r_1 \prec r_2, not\ head(r_2) \in (body(r_1))^-$ and, consequently, $(\mathcal{R}_2, \mathcal{R}_1)$ is a basic attack.

We will prove that if each derivation of $\mathcal{A}_2$ is blocked by $\mathcal{A}_1$, then it is impossible to derive the attack $(\mathcal{A}_2, \mathcal{A}_1)$.

Let the basic attack $(\mathcal{R}_2, \mathcal{R}_1)$ be given. A derivation of $(\mathcal{A}_2, \mathcal{A}_1)$ from the basic attack should contain rules Q2 or Q3 or Q4 or Q5 in order to proceed from $\mathcal{R}_2$ to $\mathcal{A}_2$ ($\mathcal{X}$ is not a basic attack). A derivation of $\mathcal{A}_2$ using R1, R2, R3 could be reconstructed from this. The derivation is blocked. Therefore, Q2, Q3, Q4 and Q5 are not applicable and the derivation of $(\mathcal{A}_2, \mathcal{A}_1)$ is impossible.

Case 3. Let be $AS(P) = \{S_1, \ldots, S_k\}, k \geq 3$. Assume that the corresponding complete argumentation structures are $\mathcal{A}_i, i = 1, \ldots, k$. Suppose that each of them is blocked. Let us denote the set $\{\mathcal{A}_i \mid i = 1, \ldots, k\}$ by O.

Suppose that the set $N \subseteq O$ contains only blocked, but not blocking complete argumentation structures (each $\mathcal{A} \in N$ is blocked and not blocking). If $O \setminus N$ contains only basic argumentation structures then the preference relation $\prec$ is cyclic. Let $M \subseteq O$ be the set of complete argumentation structures which block an argumentation structure and they are not basic argumentation structures.

We will show that there is $\mathcal{A} \in M$ which is not blocked.

We assumed to the contrary that each complete argumentation structure in M is blocked (and blocking simultaneously). If the cardinality of M is 2, Case 2 applies.

Let $\mathcal{A}_1$ be in M, i.e., $\mathcal{A}_1$ is a not basic argumentation structure. Assume (without loss of generality) that each derivation of $\mathcal{A}_1$ is blocked and $\mathcal{A}_1$ blocks a derivation of $\mathcal{A}_3$. We have to show that an attack $(\mathcal{A}_1, \mathcal{A}_3)$ is not derivable.

Consider a derivation of the attack $(\mathcal{A}_1, \mathcal{A}_3)$ and reconstruct the corresponding derivation of $\mathcal{A}_1$. Suppose that $\mathcal{A}_2$ (again without loss of generality) blocks this derivation of $\mathcal{A}_1$.

Hence, $\mathcal{A}_2$ attacks an argumentation structure $\mathcal{B}$ in the derivation of $\mathcal{A}_1$. It follows that some argumentation structure in a derivation of $\mathcal{A}_2$ attacks a basic argumentation structure in the derivation of $\mathcal{A}_1$. Consequently, neither rules Q1 and Q4, nor rules Q2 and Q3 are applicable in a derivation of the attack $(\mathcal{A}_1, \mathcal{A}_3)$. Therefore, it is not derivable.

Let $R \in \mathcal{R}$ be attacked by a warranted set Q of generating rules for some answer set of P. Since Q is warranted, there is a warranted derivation of complete argumentation structure $\mathcal{B}$ corresponding to Q. There is also a derivation of complete argumentation structure $\mathcal{A}$ corresponding to R. Q attacks R, so there is a basic argumentation structure $\mathcal{C}$ from the derivation of $\mathcal{A}$ attacked by $\mathcal{D}$ from derivation of $\mathcal{B}$. Q is warranted, rules $Q2$ and $Q3$ are applicable and hence attack $(\mathcal{D}, \mathcal{C})$ is propagated to attack $(\mathcal{B}, \mathcal{C})$. It follows that derivation of $\mathcal{A}$ is blocked.

Theorem 28 *Principle I is satisfied. Let $\mathcal{P} = (P, \prec, \mathcal{N})$ be a prioritized logic program, A_1 and A_2 be two answer sets of P. Let $R \subset P$ be a set of rules and $d_1, d_2 \in P \setminus R$ are rules, d_1 is preferred over d_2. Let A_1^+, A_2^+ be generated by the rules $R \cup \{d_1\}$ and $R \cup \{d_2\}$, respectively. Assume that each set of generating rules of $\mathcal{A}_2^+$ is attacked by a warranted set of generating rules of some answer set of P.*

Then A_2 is not a preferred answer set of $(P, \prec, \mathcal{N})$.

Proof. It is assumed that each set of generating rules of A_2 is attacked by a warranted set of generating rules of some answer set of P. A_2 is not a preferred answer set of $(P, \prec, \mathcal{N})$.

Theorem 29 *Principle IV is satisfied. Let $\mathcal{P} = (P, \prec, \mathcal{N})$ be a prioritized logic program and S be an answer set of P. Suppose that S is generated by a warranted set of rules R.*

Then S is a preferred answer set.

Proof. Let R be a set of rules generating an answer set S. If R is a warranted set of generating rules, then there is a derivation of the argumentation structure $\langle S^+ \hookleftarrow S^- \rangle$ which is warranted.

8 Conclusions

An argumentation framework has been constructed, which enables transferring attacks of rules to attacks of argumentation structures and, consequently, to warranted complete argumentation structures. Preferred answer sets correspond to warranted complete argumentation structures. This construction enables a selection of a preferred answer set whenever there is a non-empty set of standard answer sets of a program.

We did not accept the second principle from [1] and we needed to modify their first principle. On the other hand, new principles, which reflect the role of blocking, have been proposed. We stress the role of blocking – in our approach, rules can be blocked by more preferred rules, but the rules which are not blocked are handled in a declarative style.

Among goals for our future research are first of all a thorough analysis of properties and weaknesses of the presented approach (supported by an implementation of the derivation rules) and a detailed comparison to other approaches.

Finally, we have to mention the main differences between the preliminary version [11] and this paper. A more subtle set of attack derivation rules is introduced. A new assumption in Q3 ($\mathcal{A}_3$ is not attacked) changed the set of attacked derivations and, consequently, our semantics. A new and more adequate notion of warranted and blocked argumentation structure is introduced, which is based on new concepts of warranted and blocked derivations. Consequently, the notion of preferred answer set is changed. A connection of attacks between argumentation structures to different derivations of argumentation structures was not expressed in [11]. More precise and appropriate formulations of Principles IV and I are presented.

Acknowledgements: We are grateful to anonymous referees for very valuable comments and proposals. This paper was supported by the grant 1/0689/10 of VEGA.

References

1. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.
2. M. Caminada and L. Amgoud. On the evaluation of argumentation formalisms. *Artificial Intelligence*, 171(5-6):286–310, 2007.
3. J. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.
4. J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.
5. J. P. Delgrande and T. Schaub. Expressing preferences in default logic. *Artificial Intelligence*, 123(1-2):41–87, 2000.
6. Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, 170(1-2):209–244, 1996.
7. A. J. García and G. R. Simari. Defeasible logic programming: An argumentative approach. *TPLP*, 4(1-2):95–138, 2004.
8. C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.
9. T. Schaub and K. Wang. A comparative study of logic programs with preference. In *IJCAI*, pages 597–602, 2001.
10. J. Šefránek. Rethinking semantics of dynamic logic programs. In *Proc. of the Workshop NMR*, 2006.
11. J. Šefránek. Preferred answer sets supported by arguments. In *Proc. of the Workshop NMR*, 2008.
12. J. Šefránek and A. Šimko. Warranted derivation of preferred answer sets, http://kedrigern.dcs.fmph.uniba.sk/reports/. Technical Report TR-2011-027, Comenius University, Faculty of Mathematics, Physics, ans Informatics, 2011.
13. A. Šimko. Accepting the natural order of rules in a logic program with preferences. In *Proceedings of the Doctoral Consortium at ICLP 2011.*, 2011.
14. A. Šimko. Preferred answer sets - banned generating set approach. In *submitted*, 2011.
15. K. Wang, L. Zhou, and F. Lin. Alternating fixpoint theory for logic programs with priority. In *Computational Logic*, pages 164–178, 2000.

Parsing Combinatory Categorial Grammar with Answer Set Programming: Preliminary Report

Yuliya Lierler[1] and Peter Schüller[2]

[1] Department of Computer Science, University of Kentucky
yulia@cs.uky.edu
[2] Institut für Informationssysteme, Technische Universität Wien
ps@kr.tuwien.ac.at

Abstract. Combinatory categorial grammar (CCG) is a grammar formalism used for natural language parsing. CCG assigns structured lexical categories to words and uses a small set of combinatory rules to combine these categories to parse a sentence. In this work we propose and implement a new approach to CCG parsing that relies on a prominent knowledge representation formalism, answer set programming (ASP) — a declarative programming paradigm. We formulate the task of CCG parsing as a planning problem and use an ASP computational tool to compute solutions that correspond to valid parses. Compared to other approaches, there is no need to implement a specific parsing algorithm using such a declarative method. Our approach aims at producing all semantically distinct parse trees for a given sentence. From this goal, normalization and efficiency issues arise, and we deal with them by combining and extending existing strategies. We have implemented a CCG parsing tool kit — ASPCCGTK— that uses ASP as its main computational means. The C&C supertagger can be used as a preprocessor within ASPCCGTK, which allows us to achieve wide-coverage natural language parsing.

1 Introduction

The task of parsing, i.e., recovering the internal structure of sentences, is an important task in natural language processing. Combinatory categorial grammar (CCG) is a popular grammar formalism used for this task. It assigns basic and complex lexical categories to words in a sentence and uses a set of combinatory rules to combine these categories to parse the sentence. In this work we propose and implement a new approach to CCG parsing that relies on a prominent knowledge representation formalism, answer set programming (ASP) — a declarative programming paradigm. Our aim is to create a wide-coverage[3] parser which returns all semantically distinct parse trees for a given sentence.

One major challenge of natural language processing is ambiguity of natural language. For instance, many sentences have more than one plausible internal structure, which often provide different semantics to the same sentence. Consider a sentence

John saw the astronomer with the telescope.

It can denote that John used a telescope to see the astronomer, or that John saw an astronomer who had a telescope. It is not obvious which meaning is the correct one without additional context. Natural language ambiguity inspires our goal to return *all semantically distinct* parse trees for a given sentence.

CCG-based systems OPENCCG [23] and TCCG [1, 3] (implemented in the LKB toolkit) can provide multiple parse trees for a given sentence. Both use chart parsing algorithms with CCG extensions such as modalities or hierarchies of categories. While OPENCCG is primarily geared towards generating sentences from logical forms, TCCG targets parsing. However, both implementations require lexicons[4] with specialized categories. Generally, crafting a CCG lexicon is a time–consuming task. An alternative method to using a (hand-crafted) lexicon has been developed and implemented in a wide-coverage CCG parser — C&C [6, 7]. C&C relies on machine learning techniques for tagging an input sentence with CCG categories as well as for creating parse trees with a chart algorithm.As training data, C&C uses CCGbank— a corpus

[3] The goal of wide-coverage parsing is to parse sentences that are not within a controlled fragment of natural language, e.g., sentences from newspaper articles.

[4] A CCG lexicon is a mapping from each word that can occur in the input to one or more CCG categories.

of CCG derivations and dependency structures [17]. The parsing algorithm of C&C returns a *single* most probable parse tree for a given sentence.

In the approach that we describe in this paper we formulate the task of CCG parsing as a planning problem. Then we solve it using answer set programming [19, 20]. ASP is a declarative programming formalism based on the answer set semantics of logic programs [15]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. Utilizing ASP for CCG parsing allows us to control the parsing process with declarative descriptions of constraints on combinatory rule applications and parse trees. Moreover, there is no need to implement a specific parsing algorithm, as an answer set solver is used as a computational vehicle of the method. Similarly to OPENCCG and TCCG, in our ASP approach to CCG parsing we formulate a problem in such a way that multiple parse trees are computed.

An important issue inherent to CCG parsing are spurious parse trees: a given sentence may have many distinct parse trees which yield the same semantics. Various methods for eliminating such spurious parse trees have been proposed [6, 9, 24]. We adopt some of these syntactic methods in this work.

We implemented our approach in an ASPCCGTK toolkit. The toolkit equips a user with two possibilities for assigning plausible categories to words in a sentence: it can either use a given (hand-crafted) CCG lexicon or it can take advantage of the C&C supertagger [7] for this task. The second possibility provides us with wide-coverage CCG parsing capabilities comparable to C&C. The ASPCCGTK toolkit computes best-effort parses in cases where no full parse can be achieved with CCG, resulting in parse trees for as many phrases of a sentence as possible. This behavior is more robust than completely failing in producing a parse tree. It is also useful for development, debugging, and experimenting with rule sets and normalizations. In addition to producing parse trees, ASPCCGTK contains a module for visualizing CCG derivations.

A number of theoretical characterizations of CCG parsing exists. They differ in their use of specialized categories, their sets of combinatory rules, or specific conditions on applicability of rules. An ASP approach to CCG parsing implemented in ASPCCGTK can be seen as a basis of a generic tool for encoding different CCG category and rule sets in a declarative and straightforward manner. Such a tool provides a test-bed for experimenting with different theoretical CCG frameworks without the need to craft specific parsing algorithms.

The structure of this paper is as follows: we start by reviewing planning, ASP, and CCG. We describe our new approach to CCG parsing by formulating this task as a planning problem in Section 3. The implementation and framework for realizing this approach using ASP technology is the topic of Section 4. We conclude with a discussion of future work directions and challenges.

2 Preliminaries

2.1 Planning

Automated planning [5] is a widely studied area in Artificial Intelligence. In *planning*, given knowledge about

(a) available actions, their executability, and effects,
(b) an initial state, and
(c) a goal state,

the task is to find a sequence of actions that leads from the initial state to the goal state. A number of special purpose planners have been developed in this sub-area of Artificial Intelligence. Answer set programming provides a viable alternative to special-purpose planning tools [10, 18, 20].

2.2 Answer Set Programming (for Planning)

Answer set programming (ASP) [19, 20] is a declarative programming formalism based on the answer set semantics of logic programs [15, 16]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer

sets for this program. In this work we use the CLASP[5] system with its front-end (grounder) GRINGO [13], which is currently one of the most widely used answer set solvers.

A common methodology to solve a problem in ASP is to design GENERATE, DEFINE, and TEST [18] parts of a program. The GENERATE part defines a large collection of answer sets that could be seen as potential solutions. The TEST part consists of rules that eliminate the answer sets that do not correspond to solutions. The DEFINE section expresses additional concepts and connects the GENERATE and TEST parts.

A typical logic programming rule has a form of a Prolog rule. For instance, program

$$p.$$
$$q \leftarrow p,\ not\ r.$$

is composed of such rules. This program has one answer set $\{p, q\}$. In addition to Prolog rules, GRINGO also accepts rules of other kinds — "choice rules" and "constraints". For example, rule

$$\{p, q, r\}.$$

is a choice rule. Answer sets of this one-rule program are arbitrary subsets of the atoms $p,\ q,\ r$. Choice rules are typically the main members of the GENERATE part of the program. Constraints often form the TEST section of a program. Syntactically, a constraint is the rule with an empty head. It encodes the conditions on the answer sets that have to be met. For instance, the constraint

$$\leftarrow p,\ not\ q.$$

eliminates the answer sets of a program that include p and do not include q.

System GRINGO allows the user to specify large programs in a compact way, using rules with schematic variables and other abbreviations. A detailed description of its input language can be found in the online manual [13]. Grounder GRINGO takes a program "with abbreviations" as an input and produces its propositional counterpart that is then processed by CLASP. Unlike Prolog systems, the inference mechanism of CLASP is related to that of Propositional Satisfiability (SAT) solvers [14].

The GENERATE-DEFINE-TEST methodology is suitable for modeling planning problems. To illustrate how ASP programs can be used to solve such problems, we present a simplified part of the encoding of a classic toy planning domain *blocks world* given in [18]. In this domain, blocks are moved by a robot. There are a number of restrictions including the fact that a block cannot be moved unless it is clear.

Lifschitz [18] models the blocks world domain by means of five predicates: *time/1*, *block/1*, *location/1*, *move/3*, *on/3*; a location is a *block* or the *table*. The constant *maxsteps* is an upper bound on the length of a plan. States of the domain are modeled by the ground atoms of the form *on(b,l,t)* stating that block b is at location l at time t. Actions are modeled by ground atoms *move(b,l,t)* stating that block b is moved to location l at time t.

The GENERATE section of a program consists of a single rule

$$\{move(B, L, T)\} \leftarrow block(B),\ location(L),\ time(T),\ T < maxsteps.$$

that defines a potential solution to be an arbitrary set of *move* actions executed before *maxsteps*.

The fact that moving a block to a position at time T forces a block to be at this position at time $T{+}1$ is encoded in DEFINE part of the program by the rule

$$on(B, L, T{+}1) \leftarrow move(B, L, T),\ block(B),\ location(L),\ time(T),\ T{<}maxsteps.$$

The rule below specifies the commonsense law of inertia for a predicate *on* stating that unless we know that the block is no longer at the same position it remains where it was:

$$on(B, L, T{+}1) \leftarrow on(B, L, T),\ not\ \neg on(B, L, T{+}1),\ block(B),\ location(L),\ time(T),\ T < maxsteps.$$

[5] `http://potassco.sourceforge.net/`.

The following constraint in TEST encodes the restriction that a block cannot be moved unless it is clear

$$\leftarrow move(B, L, T),\ on(B1, B, T),\ block(B),\ block(B1),\\ location(L),\ time(T),\ T < maxsteps.$$

Given the rest of the encoding and the description of an initial state and of the goal state, answer sets of the resulting program represent plans. The ground atoms of the form *move(b,l,t)* present in an answer set form the list of actions of a corresponding plan.

2.3 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) [22] is a linguistic grammar formalism. Compared to other grammar formalisms, CCG uses a comparatively small set of combinatory rules – combinators – to combine comparatively rich lexical categories of words.

Categories in CCG are either atomic or complex. For instance, noun N, noun phrase NP, propositional phrase PP, and sentence S are atomic categories. Complex categories are functors that specify the type and direction of the arguments and the type of the result. A complex category

$$S\backslash NP$$

is a category for English intransitive verbs (such as *walk*, *hug*), which states that a noun phrase is required to the left, resulting in a sentence. A category

$$(S\backslash NP)/NP$$

for English transitive verbs (such as *like* and *bite*) specifies that a noun phrase is required to the right and yields the category of an English intransitive verb, which (as before) requires a noun phrase to the left to form a sentence.

Given a sentence and a lexicon containing a set of word-category pairs, we can replace words in the sentence by appropriate categories. For example, for a sentence

$$\textit{The dog bit John} \tag{1}$$

and a lexicon containing pairs

$$The \text{ - } NP/N;\ dog \text{ - } N;\ bit \text{ - } (S\backslash NP)/NP;\ John \text{ - } NP \tag{2}$$

we obtain

$$\frac{The}{NP/N} \qquad \frac{dog}{N} \qquad \frac{bit}{(S\backslash NP)/NP} \qquad \frac{John}{NP}.$$

Words may have multiple categories, e.g., “bit” is also an intransitive verb and a noun. For presentation of parsing we limit each word to one category. Our framework is able to handle multiple categories by considering all combinations of word categories.

To parse English sentences a number of combinators are required [22]: forward and backward application ($>$ and $<$, respectively), forward and backward *composition* ($>\mathbf{B}$ and $<\mathbf{B}$), forward and backward *type raising* ($>\mathbf{T}$ and $<\mathbf{T}$), backward *cross composition*, backward *cross substitution*, and *coordination*. Specifications of some of these combinators follow:

$$\frac{A/B \quad B}{A} > \qquad \frac{A/B \quad B/C}{A/C} >\mathbf{B} \qquad \frac{A}{B/(B\backslash A)} >\mathbf{T}$$

$$\frac{B \quad A\backslash B}{A} < \qquad \frac{B\backslash C \quad A\backslash B}{A\backslash C} <\mathbf{B} \qquad \frac{A}{B\backslash(B/A)} <\mathbf{T}$$

where A, B, C are variables that can be substituted by CCG categories such as N or $S\backslash NP$. An *instance* of a CCG combinator is obtained by substituting CCG categories for variables. For example,

$$\frac{NP/N \quad N}{NP} > \tag{3}$$

is an instance of the forward application combinator ($>$).

A CCG combinatory rule combines one or more adjacent categories and yields exactly one output category. To parse a sentence is to apply instances of CCG combinators so that the final category S is derived at the end. A sample CCG derivation for sentence (1) follows

$$\dfrac{\dfrac{\dfrac{\dfrac{The}{NP/N}\quad\dfrac{dog}{N}}{NP}>\quad\dfrac{\dfrac{bit}{(S\backslash NP)/NP}\quad\dfrac{John}{NP}}{S\backslash NP}>}{S}<}{}. \tag{4}$$

Section 3.1 gives a formal definition of the CCG parsing task.

Type Raising and Spurious Parses: CCG restricted to application combinators generates the same language as CCG restricted to application, composition, and type raising rules [8, 21]. One of the motivations for type raising are non-constituent coordination constructions[6] that can only be parsed with the use of raising [2, Example (2)].

Unrestricted applications of composition and type raising combinators often create spurious parse trees which are semantically equivalent to parse trees derived using application rules only. Eisner [9, Example (3)] presents a sample sentence with 12 words and 252 parses but only 2 distinct meanings. An example of a spurious parse for sentence (1) is the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{The}{NP/N}\quad\dfrac{dog}{N}}{NP}>}{S/(S\backslash NP)}\mathbf{T}\quad\dfrac{bit}{(S\backslash NP)/NP}}{S/NP}\mathbf{B}\quad\dfrac{John}{NP}}{S}> \tag{5}$$

which utilizes application, type raising, and composition combinators. Both derivations (4) and (5) have the same semantic value (in a sense, the difference between (4) and (5) is not essential for subsequent semantic analysis).

In this work we aim at the generation of parse trees that have different semantic values so that they reflect a real ambiguity of natural language, and not a spurious ambiguity that arises from the underlying CCG formalism. Various methods for dealing with spurious parses have been proposed such as limiting type raising only to certain categories [6], normalizing branching direction of consecutive composition rules by means of predictive combinators [24] or restrictions on parse tree shape [9]. We combine and extend these ideas to pose restrictions on generated parse trees within our framework. Details about normalizations and type raising limits that we implement are discussed in Section 3.3.

3 CCG Parsing via Planning

3.1 Problem Statement

We start by defining precisely the task of *CCG parsing*. We then state how this task can be seen as a planning problem.

A *sentence* is a sequence of words. An *abstract sentence representation* (ASR) is a sequence of categories annotated by a unique id. Recall that given a lexicon, we can replace words in the sentence by appropriate categories. As a result we can turn any sentence into ASR using a lexicon. For instance, for sentence (1) and lexicon (2) a sequence

$$[NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4]. \tag{6}$$

[6] E.g, in the sentence "We gave Jan a record and Jo a book", neither "Jan a record" nor "Jo a book" is a linguistic constituent of the sentence. With raising we can produce meaningful categories for these non-constituents and subsequently coordinate them using "and".

is an ASR of (1). We refer to categories annotated by id's as *annotated categories*. Members of (6) are annotated categories.

Recall that an instance of a CCG combinator C has a general form

$$\frac{X_1, \ldots, X_n}{Y}\ C.$$

We say that the sequence $[X_1, \ldots, X_n]$ is a *precondition* sequence of C, whereas Y is an *effect* of applying C. The precondition sequence and the effect of instance (3) of the combinator $>$ are $[NP/N, N]$ and NP, respectively. Given an instance C of a CCG combinator we may annotate it by (i) assigning a distinct id to each member of its precondition sequence, and (ii) assigning the id of the left most annotated category in the precondition sequence to its effect. We say that such an instance is an *annotated (combinator) instance*. For example,

$$\frac{NP/N^1 \quad N^2}{NP^1}\ > \tag{7}$$

is an annotated instance w.r.t. (3).

We say that an annotated instance C of a CCG combinator is *relevant* to an ASR sequence A if the precondition sequence of C is a substring of A. An annotated instance C is applied to an ASR sequence A by replacing the substring of A corresponding to the precondition sequence of C by its effect. For example, annotated instance (7) is relevant to ASR (6). Applying (7) to (6) yields ASR $[NP^1, (S\backslash NP)/NP^3, NP^4]$. In the following we will often say annotated combinator in place of annotated instance.

To view CCG parsing as a planning problem we need to specify states and actions of this domain. In CCG planning, states are ASRs and actions are annotated combinators. So the task is given the initial ASR, e.g., $[X_1^1, \ldots, X_n^n]$, to find a sequence of annotated combinators that leads to the goal ASR — $[S^1]$.

Let $\mathcal{C}_1$ denote annotated combinator (7), $\mathcal{C}_2$ denote

$$\frac{(S\backslash NP)/NP^3 \quad NP^4}{S\backslash NP^3}\ >,$$

and $\mathcal{C}_3$ denote

$$\frac{NP^1 \quad S\backslash NP^3}{S^1}\ >.$$

Given ASR (6) a sequence of actions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ forms a plan:

$$\begin{array}{ll}
\text{Time 0:} & [NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4] \\
\quad \text{action: } \mathcal{C}_1 & \\
\text{Time 1:} & [NP^1,\ \ (S\backslash NP)/NP^3,\ \ NP^4], \\
\quad \text{action: } \mathcal{C}_2 & \\
\text{Time 2:} & [NP^1,\ \ S\backslash NP^3], \\
\quad \text{action: } \mathcal{C}_3 & \\
\text{Time 3:} & [S^1].
\end{array} \tag{8}$$

This plan corresponds to parse tree (4) for sentence (1). On the other hand, a plan formed by a sequence of actions $\mathcal{C}_2$, $\mathcal{C}_1$, and $\mathcal{C}_3$ also corresponds to (4).

In planning the notion of *serializability* is important. Often given a plan, applying several consecutive actions in the plan in any order or in parallel does not change the effect of their application. Such plans are called *serializable*. Consequently, by allowing parallel execution of actions one may represent a class of plans by a single one. This is a well-known optimization in planning. For example, plan

$$\begin{array}{ll}
\text{Time 0:} & [NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4] \\
\quad \text{actions: } \mathcal{C}_1, \mathcal{C}_2 & \\
\text{Time 1:} & [NP^1,\ \ S\backslash NP^3], \\
\quad \text{action: } \mathcal{C}_3 & \\
\text{Time 2:} & [S^1]
\end{array}$$

may be seen as an abbreviation for a group of plans, i.e., itself, plan (8), and a plan formed by a sequence $\mathcal{C}_2$, $\mathcal{C}_1$, and $\mathcal{C}_3$. In CCG parsing as a planning problem, we are interested in finding plans of this kind, i.e., plans with concurrent actions.

We note that the planning problem that we solve is somewhat different from the one we just described as we would like to eliminate ("ban") some of the plans corresponding to spurious parses by enforcing normalizations.

3.2 ASP Encoding

In an ASP approach to CCG parsing, the goal is to encode the planning problem presented above as a logic program so that its answer sets correspond to plans. As a result answer sets of this program will contain the sequence of annotated combinators (actions, possibly concurrent) such that the application of this sequence leads from a given ASR to the ASR composed of a single category S. We present a part of the encoding `ccg.asp`[7] in the GRINGO language that solves a CCG parsing problem by means of ideas presented in Section 2.2.

First, we need to decide how we represent states — ASRs — by sets of ground atoms. To this end, we introduce symbols called "positions" that encode annotations of ASR members. In `ccg.asp`, relation $posCat(p, c, t)$ states that a category c is annotated with (position) p at time t. Relation $posAdjacent(p_L, p_R, t)$ states that a position p_L is adjacent to a position p_R at time t. In other words, a category annotated by p_L immediately precedes a category annotated by p_R in an ASR that corresponds to a state at time t (intuitively, L and R denote left and right, respectively.) These relations allow us to encode states of a CCG planning domain. For example, given an ASR (6) as the initial state, we can encode this state by the following set of facts

$$\begin{array}{l} posCat(1, rfunc(\text{“}NP\text{”}, \text{“}N\text{”}), 0).\ \ posCat(2, \text{“}N\text{”}, 0). \\ posCat(3, rfunc(lfunc(\text{“}S\text{”}, \text{“}NP\text{”}), \text{“}NP\text{”}), 0).\ \ posCat(4, \text{“}NP\text{”}, 0). \\ posAdjacent(1, 2, 0).\ \ posAdjacent(2, 3, 0).\ \ posAdjacent(3, 4, 0). \end{array} \tag{9}$$

Next we need to choose how we encode actions by ground atoms. The combinators mentioned in Section 2.3 are of two kinds: the ones whose precondition sequence consists of a single element (i.e., $>\mathbf{T}$ and $<\mathbf{T}$) and of two elements (e.g., $>$ and $<$)[8]. We call these combinators *unary* and *binary* respectively. Reification of actions is a technique used in planning that allows us to talk about common properties of actions in a compact way. To utilize this idea, we first introduce relations $unary(a)$ and $binary(a)$ for every unary and binary combinator a respectively. For a unary combinator a, a relation $occurs(a, p, c, t)$ states that a type raising action a occurring at time t raises a category identified with position p (at time t) to category c. For a binary combinator a a relation $occurs(a, p_L, p_R, t)$ states that an action a applied to positions p_L and p_R occurs at time t. For instance, given the initial state (9)

- $occurs(ruleFwdTypeR, 4, (S\backslash NP)/NP, 0)$ represents an application of the annotated combinator

$$\frac{NP^4}{(S\backslash NP)/NP^4}\ >\mathbf{T}$$

  to (9) at time 0,
- $occurs(ruleFwdAppl, 1, 2, 0)$ represents an application of (7) to (9) at time 0.

Given an atom $occurs(A, P, X, T)$ we sometimes say that an action A *modifies* a position P at time T. The GENERATE section of `ccg.asp` contains the rules of the kind

$$\begin{array}{rl} \{occurs(ruleFwdAppl, L, R, T)\} \leftarrow & posCat(L, rfunc(A, B), T),\ posCat(R, B, T), \\ & posAdjacent(L, R, T), \\ & not\ ban(ruleFwdAppl, L, T), \\ & time(T),\ T < maxsteps. \end{array}$$

[7] The complete listing of `ccg.asp` is available at `http://www.kr.tuwien.ac.at/staff/ps/aspccgtk/ccg.asp`

[8] In fact, coordination combinator is of the third type, i.e., its precondition sequence contains three elements. Presenting the details of its encoding is out of the scope of this paper.

for each combinator. Such choice rules describe a potential solution to the planning problem as an arbitrary set of actions executed before *maxsteps*. These rules also captures some of the executability conditions of the corresponding actions. For example, $posCat(L, rfunc(A, B), T)$ states that the left member of the precondition sequence of the forward application combinator $ruleFwdAppl$ is of the form A/B. At the same time, $posAdjacent(L, R, T)$ states that $ruleFwdAppl$ may be applied only to adjacent positions. A relation $ban(a, p, t)$ specifies when it is impossible for an action a to modify position p at time t. Often there are several rules defining this relation for a combinator. These rules form the main mechanism by which normalization techniques are encoded in `ccg.asp`. For instance, a rule defining ban relation

$$\begin{aligned} ban(ruleFwdAppl, L, T) \leftarrow\ & occurs(ruleBwdRaise, L, X, TLast{-}1), \\ & posLastAffected(L, TLast, T),\ time(TLast), \\ & time(T),\ T < maxsteps. \end{aligned}$$

states that a forward application modifying a position L may not *occur* at time T if the last action modifying L was backward type raising ($posLastAffected$ is an auxiliary predicate that helps to identify the last action modifying an element of the ASR). This corresponds to one of the normalization rules discussed in [9].

There are a number of rules that specify effects of actions in the CCG parsing domain. One such rule

$$\begin{aligned} posCat(L, A, T{+}1) \leftarrow\ & occurs(ruleFwdAppl, L, R, T), \\ & posCat(L, rfunc(A, B), T),\ time(T),\ T < maxsteps. \end{aligned}$$

states that an application of a forward application combinator at time T causes a category annotated by L to be X at time $T{+}1$.

The following rule characterizes an effect of binary combinators and defines the *posAffected* concept which is useful in stating several normalization conditions described in Section 3.3:

$$\begin{aligned} posAffected(L, T{+}1) \leftarrow\ & occurs(Act, L, R, T),\ binary(Act), \\ & time(T),\ T < maxsteps. \end{aligned}$$

Relation $posAffected(L, T{+}1)$ holds if the element annotated by L in the ASR was modified by a combinator at time T. Note that this rule takes advantage of reification and provides means for compact encoding of common effects of all binary actions. Furthermore, $posAffected$ is used to state the law of inertia for the predicate $posCat$

$$\begin{aligned} posCat(P, C, T{+}1) \leftarrow\ & posCat(P, C, T),\ not\ posAffected(P, T{+}1), \\ & time(T),\ T < maxsteps. \end{aligned}$$

In the TEST section of the program we encode such restrictions as no two combinators may modify the same position simultaneously and the fact that the goal has to be reached. We allow two possibilities for specifying a goal. In one case, the goal is to reach an ASR composed of a single category S by $maxsteps$. In another case, the goal is to reach the shortest possible ASR sequence by $maxsteps$.

Finally we pose additional restrictions, which ensure that only a single plan is produced when multiple serializable plans correspond to the same parse tree. Note that applying a CCG rule r at a time t creates a new category required for subsequent application of another rule r' at a time $t'{>}t$. We request that r' is applied at $t'{=}t{+}1$. Furthermore, in `ccg.asp` we enforce the condition that combinators are applied as early as possible: by requesting that a rule applied at time t uses at least one position that was modified at time $t{-}1$.

Given `ccg.asp` and the set of facts describing the initial state (ASR representation of a sentence) and the goal state (ASR containing a single category S), answer sets of the resulting program encode plans corresponding to parse trees. The ground atoms of the form $occurs(a, p, c, t)$ present in an answer set form the list of actions of a matching plan.

3.3 Normalizations

Currently, `ccg.asp` implements a number of normalization techniques and strategies for improving efficiency and eliminating spurious parses:

• One of the techniques used in C&C to improve its efficiency is to limit type raising to certain categories *based on the most commonly used type raising rule instantiations in sections 2-21 of CCGbank* [6]. We adopt this idea by limiting type raising to be applicable only to noun phrases, NP, so that NP can be raised using categories S, $S\backslash NP$, or $(S\backslash NP)/NP$. This technique reduces the size of the propositional (ground) program for `ccg.asp` and subsequently the performance of `ccg.asp` considerably. We plan to extend limiting type raising to the full set of categories used in C&C that proved to be suitable for wide-coverage parsing.
• We normalize branching direction of subsequent functional composition operations [9]. This is realized by disallowing functional forward composition to apply to a category on the left side, which has been created by functional forward composition. (And similar for backward composition.)
• We disallow certain combinations of rule applications if the same result can be achieved by other rule applications as shown in the following

$$\dfrac{\dfrac{X/Y \;\; Y/Z}{X/Z}{>\mathbf{B}} \;\; Z}{X}{>} \quad \overset{\text{normalize}}{\Rightarrow} \quad \dfrac{X/Y \;\; \dfrac{Y/Z \;\; Z}{Y}{>}}{X}{>} \qquad \dfrac{\dfrac{X}{Y/(Y\backslash X)}{>\mathbf{T}} \;\; Y\backslash X}{Y}{>} \quad \overset{\text{normalize}}{\Rightarrow} \quad \dfrac{X \;\; Y\backslash X}{Y}{<}$$

where the left-hand side is the spurious parse and the right-hand side the normalized parse. These two normalizations (plus analogous normalizations for backward composition and backward type raising) eliminate spurious parses like (5) and have been discussed in similar form in [3, 9].

4 ASPCCG Toolkit

We have implemented ASPCCGTK— a python[9] framework for using `ccg.asp`. The framework is available online[10], including documentation and examples.

Figure 1 shows a block diagram of ASPCCGTK. We use GRINGO and CLASP for ASP solving and control these solvers from python using a modified version of the BioASP library [11]. BioASP is used for calling ASP solvers as subtasks, parsing answer sets, and writing these answer sets to temporary files as facts.

Input for parsing can be (a) a natural language sentence given as a string, or (b) a sequence of words and a dictionary providing possible categories for each word, both given as ASP facts. In the first case, the framework uses C&C supertagger[11] [7] to tokenize and tag this sentence. The result of supertagging is a sequence of words of the sentence, where each word is assigned a set of likely CCG categories. From the C&C supertagger output, ASPCCGTK creates a set of ASP facts representing the sequence of words and a corresponding set of likely CCG categories. This set of facts is passed to `ccg.asp` as the initial state. In the second case (b) the input can be processed directly by `ccg.asp`. The maximum parse tree depth (i.e., the maximum plan length – *maxsteps*) currently has to be specified by the user. Auto detection of useful depth values is subject of future work.

ASPCCGTK first attempts to find a "strict" parse which requires that the resulting parse tree yields a category S (by $maxsteps$). If this is not possible, we do "best-effort" parsing using CLASP optimization features to minimize the number of categories left at the end. For instance, consider a lexicon that provides a single category for "bit", namely $(S\backslash NP)/NP$, then the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{The}}{NP/N} \;\; \dfrac{\mathit{dog}}{N}}{NP}{>} \;\; \dfrac{\mathit{bit}}{(S\backslash NP)/NP}}{S/(S\backslash NP)}{>\mathbf{T}}}{S/NP}{>\mathbf{B}} \tag{10}$$

corresponds to a best-effort parse.

[9] `http://www.python.org/`
[10] `http://www.kr.tuwien.ac.at/staff/ps/aspccgtk/`
[11] `http://svn.ask.it.usyd.edu.au/trac/candc`

Answer sets resulting from `ccg.asp` represent parse trees. ASPCCGTK passes them to a visualization component, which invokes GRINGO+CLASP on another ASP encoding `ccg2idpdraw.asp`.[12] The resulting answer sets of `ccg2idpdraw.asp` contain drawing instructions for the IDPDraw tool [25], which is used to produce a two-dimensional image for each parse tree. Figure 2 demonstrates an image generated by IDPDraw for parse tree (4) of sentence (1). If multiple parse trees exist, IDPDraw allows to switch between them.

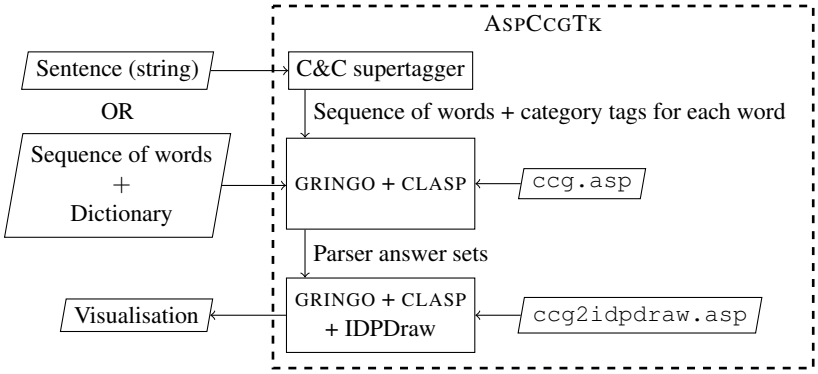


Fig. 1. Block diagram of the ASPCCG framework. (Arrows indicate data flow.)

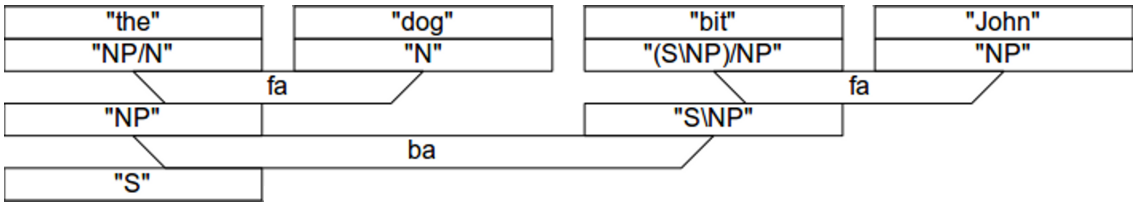


Fig. 2. Visualization of parse tree (4) for sentence (1) using IDPDraw.

5 Discussion and Future Work

Preliminary experiments on using the C&C supertagger as a front-end of ASPCCGTK yielded promising results for achieving wide-coverage parsing. The supertagger of C&C not only provides a set of likely category assignments for the words in a given sentence but also includes probability values for assigned categories. C&C uses a dynamic tagging strategy for parsing. First only very likely categories from the tagger are used for parsing. If this yields no result then less likely categories are also taken into account. In the future, we will implement a similar approach in ASPCCGTK.

We have evaluated the efficiency of ASPCCGTK on a small selection of examples from CCGbank [17]. In the future we will evaluate our parser against a larger corpus of CCGbank, considering both parsing efficiency and quality of results as evaluation criteria. Experiments done so far are encouraging and we are convinced that wide-coverage CCG parsing using ASP technology is feasible.

[12] This visualization component could be put directly into `ccg.asp`. However, for performance reasons it has proved crucial to separate the parsing calculation from the drawing calculations.

To increase parsing efficiency we consider to reformulate the CCG parsing problem as a "configuration" problem. This might improve performance. At the same time the framework would keep its beneficial declarative nature. Investigating applicability of incremental ASP [12] to enhance system's performance is another direction of future research.

Creating semantic representations for sentences is an important task in natural language processing. Boxer [4] is a tool which accomplishes this task, given a CCG parse tree from C&C. To take advantage of this advanced computational semantics tool, we aim at creating an output format for ASPCCGTK that is compatible with Boxer.

As our framework is a generic parsing framework, we can easily compare different CCG rule sets with respect to their efficiency and normalization behavior. We next discuss an idea for improving scalability of `ccg.asp` that is based on an alternative combinatory rule set to the one currently implemented in `ccg.asp`. Type raising is a core source of nondeterminism in CCG parsing and is one of the main reasons for spurious parse trees and long parsing times. In the future we would like to evaluate an approach that partially eliminates type raising by pushing it into all non-type-raising combinators. A similar strategy has been proposed for composition combinators by Wittenburg [24].[13] Combining CCG rules this way creates more combinators, however these rules contain fewer nondeterministic guesses about raising categories. The reduced nondeterminism should improve solving efficiency without losing any CCG derivations.

Acknowledgments. We would like to thank John Beavers and Vladimir Lifschitz for valuable detailed comments on the first draft of this paper. We are grateful to Jason Baldridge, Johan Bos, Esra Erdem, Michael Fink, Michael Gelfond, Joohyung Lee, and Miroslaw Truszczynski for useful discussions related to the topic of this work, as well as to the anonymous reviewers for their feedback. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship. Peter Schüller was supported by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

References

1. Beavers, J.: Documentation: A CCG implementation for the LKB. Tech. rep., Stanford University, Center for the Study of Language and Information (2003)
2. Beavers, J., Sag, I.: Coordinate ellipsis and apparent non-constituent coordination. In: International Conference on Head-Driven Phrase Structure Grammar (HPSG'04). pp. 48–69 (2004)
3. Beavers, J.: Type-inheritance combinatory categorial grammar. In: International Conference on Computational Linguistics (COLING'04) (2004)
4. Bos, J.: Wide-coverage semantic analysis with boxer. In: Bos, J., Delmonte, R. (eds.) Semantics in Text Processing. STEP 2008 Conference Proceedings. pp. 277–286. Research in Computational Semantics, College Publications (2008)
5. Cimatti, A., Pistore, M., Traverso, P.: Automated planning. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation. Elsevier (2008)
6. Clark, S., Curran, J.R.: Log-linear models for wide-coverage CCG parsing. In: SIGDAT Conference on Empirical Methods in Natural Language Processing (EMNLP-03) (2003)
7. Clark, S., Curran, J.R.: Parsing the WSJ using CCG and log-linear models. In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL'04). pp. 104–111. Barcelona, Spain (2004)
8. Dowty, D.: Type raising, functional composition, and non-constituent conjunction. In: Oehrle, R.T., Bach, E., Wheeler, D. (eds.) Categorial grammars and natural language structures, vol. 32, pp. 153–197. Dordrecht, Reidel (1988)
9. Eisner, J.: Efficient normal-form parsing for combinatory categorial grammar. In: Proceedings of the 34th annual meeting on Association for Computational Linguistics (ACL'96). pp. 79–86 (1996)
10. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. ACM Trans. Comput. Logic 5, 206–263 (April 2004)
11. Gebser, M., König, A., Schaub, T., Thiele, S., Veber, P.: The BioASP library: ASP solutions for systems biology. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10). vol. 1, pp. 383–389 (2010)

[13] Wittenburg introduced a new set of combinatory rules by combining the functional composition combinators with other combinators. By omitting the original functional composition combinators, certain spurious parse trees can no longer be derived.

12. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental asp solver. In: Proceedings of International Logic Programming Conference and Symposium (ICLP'08) (2008)
13. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. (2010), http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/guide.pdf
14. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium (ICLP'88). pp. 1070–1080. MIT Press (1988)
16. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
17. Hockenmaier, J., Steedman, M.: CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. Comput. Linguist. 33, 355–396 (2007)
18. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
19. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
20. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
21. Partee, B., Rooth, M.: Generalized conjunction and type ambiguity. In: Baeuerle, R., Schwarze, C., von Stechov, A. (eds.) Meaning, Use, and Interpretation. pp. 361–383 (1983)
22. Steedman, M.: The syntactic process. MIT Press, London (2000)
23. White, M., Baldridge, J.: Adapting chart realization to CCG. In: European Workshop on Natural Language Generation (EWNLG'03) (2003)
24. Wittenburg, K.: Predictive combinators: a method for efficient processing of combinatory categorial grammars. In: 25th Annual Meeting of the Association for Computational Linguistics (ACL'87). pp. 73–80 (1987)
25. Wittocx, J.: IDPDraw (2009), Katholieke Universiteit Leuven, http://dtai.cs.kuleuven.be/krr/software/download

Solving Modular Model Expansion Tasks

Shahab Tasharrofi, Xiongnan (Newman) Wu, Eugenia Ternovska

Simon Fraser University
{sta44,xwa33,ter}@cs.sfu.ca

Abstract. The work we describe here is a part of a research program of developing foundations of declarative solving of search problems. We consider the model expansion task as the task representing the essence of search problems where we are given an instance of a problem and are searching for a solution satisfying certain properties. Such tasks are common in artificial intelligence, formal verification, computational biology. Recently, the model expansion framework was extended to deal with multiple modules. In the current paper, inspired by practical combined solvers, we introduce an algorithm to solve model expansion tasks for modular systems. We show that our algorithm closely corresponds to what is done in practice in different areas such as Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), Answer Set Programming (ASP).

1 Introduction

The research described in this paper is a part of a research program of developing formal foundations for specification/modelling languages (declarative programming) for solving computationally hard problems. In [1], the authors formalize search problems as the logical task of *model expansion (MX)*, the task of expanding a given (mathematical) structure with new relations. They started a research program of finding common underlying principles of various approaches to specifying and solving search problems, finding appropriate mathematical abstractions, and investigating complexity-theoretic and expressiveness issues. It was emphasized that it is important to understand the expressiveness of a specification language in terms of the computational complexity of the problems it can represent. Complexity-theoretic aspects of model expansion for several logics in the context of related computational tasks of satisfiability and model checking were studied in [2]. Since built-in arithmetic is present in all realistic modelling languages, it was important to formalize built-in arithmetic in such languages. In [3], model expansion ideas were extended to provide mathematical foundation for dealing with arithmetic and aggregate functions (min, sum etc.). There, the instance and expansion structures are embedded into an infinite structure of arithmetic, and the property of capturing NP was proven for a logic which corresponds to practical languages. The proposed formalism applies to other infinite background structures besides arithmetic. The analysis of practical languages was given in [4]. It was proved that certain common problems involving numbers (e.g. integer factorization) are not expressible in the ASP and IDP system languages naturally, and in [5], the authors improved the result of [3] by defining a new logic which unconditionally captures NP over arithmetical structures.

The next step in the development of the MX-based framework is adding modularity concepts. It is convenient from the point of view of a user to be able to split a large problem into subproblems, and to use the most suitable formalism for each part, and thus a unifying semantics is needed. In a recent work [6], a subset of the authors extended the MX framework to be able to represent a modular system. The most interesting aspect of that proposal is that modules can be considered from both model-theoretic and operational view. Under the model-theoretic view, an MX module is a set (or class) of structures, and under the operational view it is an operator, mapping a subset of the vocabulary to another subset. An abstract algebra on MX modules is given, and it allows one to combine modules on abstract model-theoretic level, independently from what languages are used for describing them. Perhaps the most important operation in the algebra is the loop (or feedback) operation, since iteration underlies many solving methods. The authors show that the power of the loop operator is such that the combined modular system can capture all of the complexity class NP even when each module is deterministic and polytime. Moreover, in general, adding loops gives a jump in the polynomial time hierarchy, one step from the highest complexity of the components. It is also shown that each module can be viewed as an operator, and when each module is

(anti-) monotone, the number of the potential solutions can be significantly reduced by using ideas from the logic programming community.

To develop the framework further, we need a method for "solving" modular MX systems. By solving we mean finding structures which are in the modular system, where the system is viewed as a function of individual modules. *Our goal is to come up with a general algorithm which takes a modular system in input and generates its solutions.*

We take our inspiration in how "combined" solvers are constructed in the general field of declarative problem solving. The field consists of many areas such as Integer Linear Programming (ILP), Answer Set Programming (ASP), Satisfiability Modulo Theories (SMT), Satisfiability (SAT), and Constraint Programming (CP), and each of these areas has developed multitudes of solvers, including powerful "combined" solvers such as SMT solvers. Moreover, SMT-like techniques are needed in the ASP community [7]. Our main challenge is to come up with an appropriate mathematical abstraction of "combined" solving. Our contributions are as follows.

1. We formalize common principles of "combined" solving in different communities in the context of modular model expansion. Just as in [6], we use a combination of a model-theoretic, algebraic and operational view of modular systems.
2. We design an abstract algorithm that given a modular system, computes the models of that modular system iteratively, and we formulate conditions on languages of individual modules to participate in the iterative solving. We use the formalization above of these common principles to show the effectiveness of our algorithm.
3. We introduce abstractions for many ideas in practical systems such as the concept of a *valid acceptance procedure* that abstractly represents unit propagation in SAT, well-founded model computation in ASP, arc-consistency checkers in CP, etc.
4. As a proof of concept, we show that, in the context of the model expansion task, our algorithm generalizes the work of different solvers from different communities in a unifying and abstract way. In particular, we investigate the branch-and-cut technique in ILP and methods used in SMT, DPLL(Agg) and combinations of ASP and CP [8–11]. We aim to show that, although no implementation is presented, the algorithm should work fine as it mimics the current technology.
5. We develop an improvement of our algorithm by using approximation methods proposed in [6].

2 Background

2.1 Model Expansion

In [1], the authors formalize combinatorial search problems as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes the problem in some logic $\mathcal{L}$. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification/modelling language. It could be an extension of first-order logic such as FO(ID), or an ASP language, or a modelling language from the CP community such as ESSENCE [12].

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions. The task of model expansion for an arbitrary logic $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$
Given: *1.* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$
2. A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

Thus, we expand the structure $\mathcal{A}$ with relations and functions to interpret ε, obtaining a model B of ϕ. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[1].

Example 1. The following formula ϕ of first order logic constitutes an MX specification for Graph 3-colouring:

$$\begin{gathered}\forall x\,[(R(x) \vee B(x) \vee G(x)) \\ \wedge\neg((R(x) \wedge B(x)) \vee (R(x) \wedge G(x)) \vee (B(x) \wedge G(x)))] \\ \wedge\, \forall x \forall y\,[E(x,y) \supset (\neg(R(x) \wedge R(y)) \\ \wedge\neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))].\end{gathered}$$

An instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathcal{A} = \mathcal{G} = (V; E)$. The task is to find an interpretation for the symbols of the expansion vocabulary $\varepsilon = \{R, B, G\}$ such that the expansion of $\mathcal{A}$ with these is a model of ϕ:

$$\underbrace{(\overbrace{V; E^{\mathcal{A}}}^{\mathcal{A}},\ R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

The interpretations of ε, for structures $\mathcal{B}$ that satisfy ϕ, are exactly the proper 3-colourings of $\mathcal{G}$.

Given a specification, we can talk about a set of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, we can simply talk about a set of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy. This abstract view makes our study of modularity language-independent.

2.2 Modular Systems

This section reviews the concept of a modular system defined in [6] based on the initial development in [13]. As in [6], *each modular system abstractly represents an MX task*, i.e., a set (or class) of structures over some instance and expansion vocabulary. A modular system is formally described as a set of primitive modules (individual MX tasks) combined using the operations of:

1. Projection($\pi_\tau(M)$) which restricts the vocabulary of a module,
2. Composition($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2,
3. Union($M_1 \cup M_2$),
4. Feedback($M[R = S]$) which connects output S of M to its inputs R and,
5. Intersection($M_1 \cap M_2$).

Formal definitions of these operations are not essential for understanding this paper, thus, we refer the reader to [6] for details. The algebraic operations are illustrated in Examples 2 and 3. In this paper, we only consider modular systems which do not use the union operator.

Our goal in this paper is to solve the MX task for a given modular system, i.e., given a modular system M (described in algebraic terms using the operations above) and structure $\mathcal{A}$, find a structure $\mathcal{B}$ in M expanding $\mathcal{A}$. We find our inspiration in existing solver architectures by viewing them at a high level of abstraction.

Example 2 (Timetabling [13]). Here, we use the example of timetabling from [13] and modify its representation using our additional feedback operator. Figure 1 shows the new modular representation of the timetabling problem where the event data and the resource data are the inputs and a list of events with their associated sessions and resources (locations) is the output. This timetabling is done so that the allocations of resource and sessions to the events do not conflict. Unlike [13] where the "allDifferent" module is completely independent of the "testAllocation" module, here, through our feedback operator, these modules are inter-dependent. This inter-dependency provides a better model of the whole system by making the model closer to the reality. Also, here, unlike [13] module "allDifferent" can be a deterministic module. In fact, as proved in [6], the non-determinacy of all NP problems can be modeled through the feedback operator. As will be shown later in this paper, the existence of such loops can also help us to speed up the solving process of some problems.

[1] By ":=" we mean "is by definition" or "denotes".

In this paper, we propose an algorithm such that: given a modular system as in Figure 1 and given the inputs to this modular system, the algorithm finds a solution to (or a model of) the given modular system, i.e., an interpretation to the symbol "occurs" on this example that is not in conflict with the constraints on the timetable.

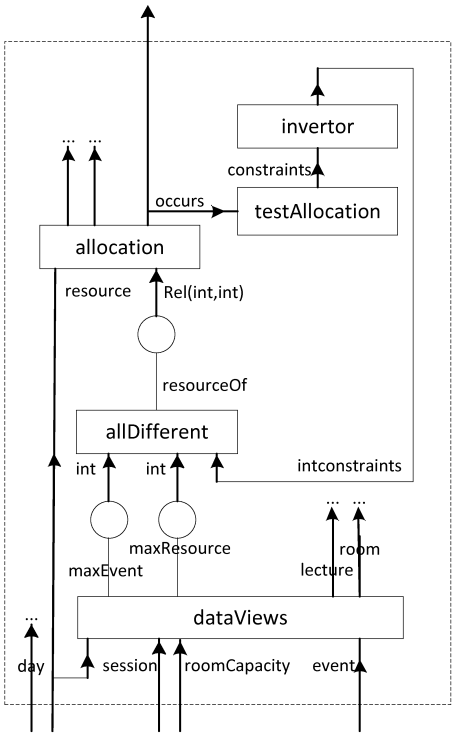


Fig. 1. Modular System Representing a Timetabling Problem

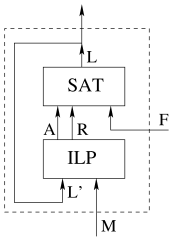


Fig. 2. Modular System Representing an SMT Solver for the Theory of Integer Linear Arithmetic

Example 3 (SMT Solvers). Consider Figure 2: The inner boxes (with solid borders) show simpler MX modules and the outer box shows our module of interest. The vocabulary consists of all symbols A, R, L, L', M and F where A, R and L' are internal to the module, and others form its interface. Also, there is a feedback from L to L'.

Overall, this modular system describes a simple SMT solver for the theory of Integer Linear Arithmetic (T_{ILA}). Our two MX modules are SAT and ILP. They work on different parts of a specification. The ILP module takes a set L' of literals and a mapping M from atoms to linear arithmetic formulas. It returns two sets R and A. Semantically, R represents a set of subsets of L' so that $T_{ILA} \cup M|_r$[2] is *unsatisfiable* for all subsets $r \in R$. Set A represents a set of propagated literals together with their justifications, i.e., a set of pairs (l, Q) where l is an unassigned literal (i.e., neither $l \in L'$ nor $\neg l \in L'$) and Q is a set of assigned literals asserting $l \in L'$, i.e., $Q \subseteq L'$ and $T_{ILA} \cup M|_Q \models M|_l$ (the ILA formula $M|_l$ is a logical consequence of ILA formulas $M|_Q$). The SAT module takes R and A and a propositional formula F and returns set L of literals such that: (1) L makes F true, (2) L is not a superset of any $r \in R$ and, (3) L respects all propagations (l, Q) in A, i.e., if $Q \subseteq L$ then $l \in L$. Using these modules and our operators, module SMT is defined as below to represent our simple SMT solver:

$$SMT := \pi_{\{F,M,L\}}((ILP \rhd SAT)[L = L']). \tag{1}$$

The combined module SMT is correct because, semantically, L satisfies F and all models in it should have $R = \emptyset$, i.e., $T_{ILA} \cup M|_L$ is satisfiable. This is because ILP contains structures for which if $r \in R$, then $r \subseteq L' = L$. Also, for structures in SAT, if $r \in R$ then $r \not\subseteq L$. Thus, to satisfy both these conditions, R has to be empty. Also, one can easily see that all sets L which satisfy F and make $T_{ILA} \cup M|_L$ satisfiable are solutions to this modular system (set $A = R = \emptyset$ and $L' = L$).

So, there is a one-to-one correspondence between models of the modular system above and SMT's solutions to the propositional part. To find a solution, one can compute a model of this modular system. Note that, looking at modules as operators, all models of module SMT are its fixpoints.

A description of a modular system (1) looks like a formula in some logic. One can define a satisfaction relation for that logic, however it is not needed here. Still, since each modular system is a set of structures, we call the structures in a modular system *models* of that system. We are looking for models of a modular system M which expand a given instance structure $\mathcal{A}$. We call them *solutions of M for $\mathcal{A}$*.

3 Computing Models of Modular Systems

In this section, we introduce an algorithm which takes a modular system M and a structure $\mathcal{A}$ and finds an expansion $\mathcal{B}$ of $\mathcal{A}$ in M. Our algorithm uses a tool external to the modular system (a solver). It uses modules of a modular system to "assist" the solver in finding a model (if one exists). Starting from an empty expansion of $\mathcal{A}$ (i.e., a partial structure which contains no information about the expansion predicates), the solver gradually extends the current structure (through an interaction with the modules of the given modular system) until it either finds a model that satisfies the modular system or concludes that none exists. To model this procedure, a definition of a partial structure is needed.

3.1 Partial Structures

Recall that a structure is a domain together with an interpretation of a vocabulary. A partial structure, however, may contain unknown values. For example, for a structure $\mathcal{B}$ and a unary relation R, we may know that $\langle 0 \rangle \in R^{\mathcal{B}}$ and $\langle 1 \rangle \notin R^{\mathcal{B}}$, but we may not know whether $\langle 2 \rangle \in R^{\mathcal{B}}$ or $\langle 2 \rangle \notin R^{\mathcal{B}}$. Partial structures deal with gradual accumulation of knowledge.

Definition 1 (Partial Structure). *We say $\mathcal{B}$ is a τ_p-partial structure over vocabulary τ if:*

1. *$\tau_p \subseteq \tau$,*
2. *$\mathcal{B}$ gives a total interpretation to symbols in $\tau \backslash \tau_p$ and,*
3. *for each n-ary symbol R in τ_p, $\mathcal{B}$ interprets R using two sets R^+ and R^- such that $R^+ \cap R^- = \emptyset$, and $R^+ \cup R^- \subsetneq (dom(\mathcal{B}))^n$.*

[2] For a set τ of literals, $M|_\tau$ denotes a set of linear arithmetical formulas containing: (1) M's image of positive atoms in τ, and (2) the negation of M's image of negative atoms in τ.

We say that τ_p is the partial vocabulary of $\mathcal{B}$. If $\tau_p = \emptyset$, then we say $\mathcal{B}$ is total. For two partial structures $\mathcal{B}$ and $\mathcal{B}'$ over the same vocabulary and domain, we say that $\mathcal{B}'$ extends *$\mathcal{B}$ if all unknowns in $\mathcal{B}'$ are also unknowns in $\mathcal{B}$, i.e., $\mathcal{B}'$ has at least as much information as $\mathcal{B}$.*

If a partial structure $\mathcal{B}$ has enough information to satisfy or falsify a formula ϕ, then we say $\mathcal{B} \models \phi$, or $\mathcal{B} \models \neg\phi$, respectively. Note that for partial structures, $\mathcal{B} \models \neg\phi$ and $\mathcal{B} \not\models \phi$ may be different. We call a ε-partial structure $\mathcal{B}$ over $\sigma \cup \varepsilon$ the *empty expansion* of σ-structure $\mathcal{A}$, if $\mathcal{B}$ agrees with $\mathcal{A}$ over σ but $R^+ = R^- = \emptyset$ for all $R \in \varepsilon$.

In the following, by structure we always mean a total structure, unless otherwise specified. We may talk about "bad" partial structures which, informally, are the ones that cannot be extended to a structure in M. Also, when we talk about a τ_p-partial structure, in the MX context, τ_p is always a subset of ε.

Total structures are partial structures with no unknown values. Thus, in the algorithmic sense, total structures need no further guessing and should only be checked against the modular system. A good algorithm rejects "bad" partial structures sooner, i.e., the sooner a "bad" partial structure is detected, the faster the algorithm is.

Up to now, we defined partial and total structures and talked about modules rejecting "bad" partial structures. However, modules are sets of structures (in contrast with sets of partial structures). Thus, acceptance of a partial structure has to be defined properly. Towards this goal, we first formalize the informal concept of "good" partial structures. The actual acceptance procedure for partial structures is defined later in the section.

Definition 2 (Good Partial Structures). *For a set of structures S and partial structure $\mathcal{B}$, we say $\mathcal{B}$ is a* good partial structure wrt *S if there is $\mathcal{B}' \in S$ which extends $\mathcal{B}$.*

3.2 Requirements on the Modules

Untill now, we have the concept of partial structures that the solver can work on, but, clearly, as the solver does not have any information about the internals of the modules, it needs to be assisted by the modules. Therefore, the next question could be: "what assistance does the solver need from modules so that its correctness is always guaranteed?" Intuitively, modules should be able to tell whether the solver is on the "right" direction or not, i.e., whether the current partial structure is bad, and if so, tell the solver to stop developing this direction further. We accomplish this goal by letting a module accept or reject a partial structure produced by the solver and, in the case of rejection, provide a "reason" to prevent the solver from producing the same model later on. Furthermore, a module may "know" some extra information that solver does not. Due to this fact, modules may give the solver some hints to accelerate the computation in the current direction. Our algorithm models such hints using "advices" to the solver.

Note that reasons and advices we are now talking about are different from predicate symbols R and A in Example 3. While, conceptually, R and A also represent reasons and advices there, to our algorithm, they are just predicate symbols for which an interpretation has to be found. On the other hand, reasons and advices used by our algorithm are not specific to a modular system. They are entities known to our algorithm which contain information to guide the solver in its search for a model.

Also note that in order to pass a reason or an advice to a solver, there should be a common language that the solver and the modules understand (although it may be different from all internal languages of the modules). We expect this language to have its own model theory and to support basic syntax such as conditionals or negations. We expect the model theory of this language to 1) be monotone: adding a sentence can not decrease the set of consequences and 2) have resolution theorem which is the converse of the deduction theorem, i.e., if $\Gamma \models A \supset B$ then $\Gamma \cup \{A\} \models B$. The presence of the resolution theorem guarantees that, once an advice of form $Pre \supset Post$ is added to the solver, and when the solver has deduced Pre under some assumptions, it can also deduce $Post$ under the same assumptions. From now on, we assume that our advices and reasons are expressed in such a language.

We talked about modules assisting the solver, but a module is a set of structures and has no computational power. Instead, we associate each module with an "oracle" to accept/reject a partial structure and give "reasons" and "advices" accordingly. Note that it is unreasonable to require a strong acceptance condition from oracles because, for example, assuming access to oracles which accept a partial structure iff it

is a good partial structure, one can always find a total model by polynomially many queries to such oracles. While theoretically possible, in practice, access to such oracles is usually not provided. Thus, we have to (carefully) relax our assumptions for a weaker procedure (what we call a Valid Acceptance Procedure).

Definition 3 (Advice). *Let Pre and $Post$ be formulas in the common language of advices and reasons, Formula $\phi := Pre \supset Post$ is an* advice *wrt a partial structure $\mathcal{B}$ and a set of structures M if:*

1. *$\mathcal{B} \models Pre$,*
2. *$\mathcal{B} \not\models Post$ and,*
3. *for every total structure $\mathcal{B}'$ in M, we have $\mathcal{B}' \models \phi$.*

The role of an advice is to prune the search and to accelerate extending a partial structure $\mathcal{B}$ by giving a formula that is not yet satisfied by $\mathcal{B}$, but is always satisfied by any total extensions of $\mathcal{B}$ in M. Pre corresponds to the part that is satisfied by $\mathcal{B}$ and $Post$ corresponds to the unknown part that is not yet satisfied by $\mathcal{B}$.

Definition 4 (Valid Acceptance Procedure). *Let S be a set of τ-structures. We say that P is a valid acceptance procedure for S if for all τ_p-partial structures $\mathcal{B}$, we have:*

- *If $\mathcal{B}$ is total, then if $\mathcal{B} \in S$, then P accepts $\mathcal{B}$, and if $\mathcal{B} \notin S$, then P rejects $\mathcal{B}$.*
- *If $\mathcal{B}$ is not total but $\mathcal{B}$ is good wrt S, then P accepts $\mathcal{B}$.*
- *If $\mathcal{B}$ is neither total nor good wrt $\mathcal{B}$, then P is free to either accept or reject $\mathcal{B}$.*

The procedure above is called valid as it never rejects any good partial structures. However, it is a weak acceptance procedure because it may accept some bad partial structures. This kind of weak acceptance procedures are abundant in practice, e.g., Unit Propagation in SAT, Arc-Consistency Checks in CP, and computation of Founded and Unfounded Sets in ASP. As these examples show, such weak notions of acceptance can usually be implemented efficiently as they only look for local inconsistencies. Informally, oracles accept/reject a partial structure through a valid acceptance procedure for a set containing all possible instances of a problem and their solutions. We call this set a Certificate Set.

In theoretical computer science, a problem is a subset of $\{0,1\}^*$. In logic, a problem corresponds to a set of structures. Here, we use the logic notion.

Definition 5 (Certificate Set). *Let σ and ε be instance and expansion vocabularies. Let $\mathcal{P}$ be a problem, i.e., a set of σ-structures, and C be a set of $(\sigma \cup \varepsilon)$-structures. Then, C is a* $(\sigma \cup \varepsilon)$-certificate set for $\mathcal{P}$ *if for all σ-structures $\mathcal{A}$: $\mathcal{A} \in \mathcal{P}$ iff there is a structure $\mathcal{B} \in C$ that expands $\mathcal{A}$.*

Oracles are the interfaces between our algorithm and our modules. Next we present conditions that oracles should satisfy so that their corresponding modules can contribute to our algorithm.

Definition 6 (Oracle Properties). *Let $\mathcal{L}$ be a formalism with our desired properties. Let $\mathcal{P}$ be a problem, and let O be an oracle. We say that O is*

- Complete and Constructive (CC) wrt $\mathcal{L}$ *if O returns a reason $\psi_{\mathcal{B}}$ in $\mathcal{L}$ for each partial structure $\mathcal{B}$ that it rejects such that: (1) $\mathcal{B} \models \neg\psi_{\mathcal{B}}$ and, (2) all total structures accepted by O satisfy $\psi_{\mathcal{B}}$.*
- Advising (A) wrt $\mathcal{L}$ *if O provides a set of advices in $\mathcal{L}$ wrt $\mathcal{B}$ for all partial structures $\mathcal{B}$.*
- Verifying (V) *if O is a valid acceptance procedure for some certificate set C for P.*

Oracle O is *complete wrt $\mathcal{L}$* because it ensures the existence of such a sentence and *constructive* because it provides such a sentence. Oracle O differs from the usual oracles in the sense that it does not only give yes/no answers, but also provides reasons for why the answer is correct. It is *advising* because it provides some facts that were previously unknown to guide the search. Finally, it is *verifying* because it guides the partial structure to a solution through a valid acceptance procedure. Although the procedure can be weak as described above, good partial structures are never rejected and O always accepts or rejects total structures correctly. This property guarantees the convergence to a total model. In the following sections, we use the term CCAV oracle to denote an oracle which is complete, constructive, advising, and verifying.

3.3 Requirements on the Solver

In this section, we discuss properties that a solver needs to satisfy. Although the solver can be realized by many practical systems, for them to work in an orderly fashion and for algorithm to converge to a solution fast, it has to satisfy certain properties. First, the solver has to be online since the oracles keep adding reasons and advices to it. Furthermore, to guarantee the termination, the solver has to guarantee progress, which means it either reports a proper extension of the previous partial structure or, if not, the solver is guaranteed to never return any extension of that previous partial structure later on. Now, we give the requirements on the solver formally.

Definition 7 (Complete Online Solver). *A solver S is* complete and online *if the following conditions are satisfied by S:*

- *S supports the actions of initialization, adding sentences, and reporting its state as either $\langle UNSAT\rangle$ or $\langle SAT, \mathcal{B}\rangle$.*
- *If S reports $\langle UNSAT\rangle$ then the set of sentences added to S are unsatisfiable,*
- *If S reports $\langle SAT, \mathcal{B}\rangle$ then $\mathcal{B}$ does not falsify any of the sentences added to S,*
- *If S has reported $\langle SAT, \mathcal{B}_1\rangle, \cdots, \langle SAT, \mathcal{B}_n\rangle$ and $1 \leq i < j \leq n$, then either $\mathcal{B}_j$ is a proper extension of $\mathcal{B}_i$ or, for all $k \geq j$, $\mathcal{B}_k$ does not extend $\mathcal{B}_i$.*

A solver as above is guaranteed to be sound (it returns partial structures that at least do not falsify any of the constraints) and complete (it reports unsatisfiability only when unsatisfiability is detected and not when, for example, some heuristic has failed to find an answer or some time limit is reached). Also, for finite structures, such a solver guarantees that our algorithm either reports unsatisfiability or finds a solution to modular system M and instance structure $\mathcal{A}$.

3.4 Lazy Model Expansion Algorithm

In this section, we present an iterative algorithm to solve model expansion tasks for modular systems. Algorithm 1 takes an instance structure and a modular system (and its CCAV oracles) and integrates them with a complete online solver to iteratively solve a model expansion task. The algorithm works by accumulating reasons and advices from oracles and gradually converging to a solution to the problem.

The role of the reasons is to prevent some bad structures and their extensions from being proposed more than once, i.e., when a model is deducted to be bad by an oracle, a new reason is provided by the oracle and added to the solver such that all models of the system satisfy that reason but the "bad" structure does not. The role of an advice is to provide useful information to the solver (satisfied by all models) but not yet satisfied by partial structure $\mathcal{B}$. Informally, an advice is in form "if Pre then Post", where "Pre" corresponds to something already satisfied by current partial structure $\mathcal{B}$ and "Post" is something that is always satisfied by all models of the modular system satisfying the "Pre" part, but not yet satisfied by partial structure $\mathcal{B}$. It essentially tells the solver that "Post" part is satisfied by all intended structures (models of the system) extending $\mathcal{B}$, thus helping the solver to accelerate its computation in its current direction.

The role of the solver is to provide a possibly good partial structure to the oracles, and if none of the oracles rejects the partial structure, keep extending it until we find a solution or conclude none exists. If the partial structure is rejected by any one of the oracles, the solver gets a reason from the oracle for the rejection and tries some other partial structures. The solver also gets advices from oracles to accelerate the search.

4 Examples: Modelling Existing Frameworks

In this section, we describe algorithms from three different areas and show that they can be effectively modelled by our proposed algorithm in the context of model expansion. Note that our purpose here is not to analyze other systems but to show the effectiveness of our algorithm in the absence of an implementation. We establish this claim by showing that our algorithm acts similar to the state-of-the-art algorithms when the right components are provided.

```
Data: Modular System M with each module M_i associated with a CCAV oracle O_i, input structure A and
      complete online solver S
Result: Structure B that expands A and is in M
begin
    Initialize the solver S using the empty expansion of A ;
    while TRUE do
        Let R be the state of S ;
        if R = ⟨UNSAT⟩ then return Unsatisfiable ;
        else if R = ⟨SAT, B⟩ then
            Add the set of advices from oracles wrt B to S ;
            if M does not accept B then
                Find a module M_i in M such that M_i does not accept B|_{vocab(M_i)} ;
                Let ψ be the reason given by oracle O_i ;
                Add ψ to S ;
            else if B is total then return B ;
end
```

Algorithm 1: Lazy Model Expansion Algorithm

4.1 Modelling DPLL(T)

DPLL(T) [14] system is an abstract framework to model the lazy SMT approach. It is based on a general DPLL(X) engine, where X can be instantiated with a theory T solver. DPLL(T) engine extends the Decide, UnitPropagate, Backjump, Fail and Restart actions of the classic DPLL framework with three new actions: (1) **TheoryPropagate** gives literals that are T-consequences of current partial assignment, (2) T**-Learn** learns T-consistent clauses, and (3) T**-Forget** forgets some previous lemmas of theory solver.

To participate in DPLL(T) solving architecture, a theory solver provides three operations: (1) taking literals that have been set true, (2) checking if setting these literals true is T-consistent and, if not, providing a subset of them that causes inconsistency, (3) identifying some currently undefined literals that are T-consequences of current partial assignment and providing a justification for each. More details can be found in [14].

The modular system $DPLL(T)$ of the DPLL(T) system is the same as the one in Example 3, except that we have module M_P instead of SAT and M_T instead of ILP. In Figure 2, A corresponds to the result of TheoryPropagate action that contains some information about currently undefined values in L' together with their justifications; R is calculated from the T-Learn action and corresponds to reasons of M_T rejecting L'.

To model DPLL(T), we introduce a solver S to be any DPLL-based online SAT solver, so that it performs the basic actions of Decide, UnitPropagate, Fail, Restart, and also Backjump when the backjumping clause is added the solver. The two modules M_T and M_P are attached with oracles O_T and O_P respectively. They accept a partial structure $\mathcal{B}$ iff their respective module constraints is not falsified by $\mathcal{B}$. When rejecting $\mathcal{B}$, a reason "P_{in} then P_{out}" (true about all models of the module) is returned where P_{in} (resp. P_{out}) is a property about input (resp. output) vocabulary of the module satisfied (resp. falsified) by $\mathcal{B}$. They may also return advices of the same form but with P_{out} being neither satisfied nor falsified by $\mathcal{B}$. The constructions of these two modules are similar; so, we only give a construction for the solver S and module M_T:

Solver S is a DPLL-based online SAT solver (clearly complete and online).

Module M_T The associated oracle O_T accepts a partial structure $\mathcal{B}$ if it does not falsify the constraints described in Example 3 on L', M, A, and R for module M_T. If $\mathcal{B}$ is rejected, O_T returns a reason $\psi := \psi_{in} \supset \psi_{out}$ where $\mathcal{B}|_{\{L',M\}} \models \psi_{in}$ but $\mathcal{B}|_{\{A,R\}} \models \neg\psi_{out}$. Clearly, $\mathcal{B} \models \neg\psi$ and all models in M_T satisfy ψ. Thus, O_T is complete and constructive. O_T may also return some advices which are similar to the reason above except that ψ_{out} is neither satisfied nor falsified by $\mathcal{B}$. Hence, O_T is an advising oracle. Also, O_T always makes the correct decision for a total structure and rejects a partial structure only when it falsifies the constraints for M_T. O_T never rejects any good partial structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_T is a valid acceptance procedure for M_T and, thus, a verifying oracle.

Proposition 1. *1. Modular system $DPLL(T)$ models the DPLL(T) system. 2. Solver S is complete and online. 3. O_P and O_T are CCAV oracles.*

DPLL(T) architecture is known to be very efficient and many solvers are designed to use it, including most SMT solvers [9]. The DPLL(Agg) module [10] is suitable for all DPLL-based SAT, SMT and ASP solvers to check satisfiability of aggregate expressions in DPLL(T) context. All these systems are representable in our modular framework.

4.2 Modelling ILP Solvers

Integer Linear Programming solvers solve optimization problems. In this paper, we model ILP solvers which use general branch-and-cut method to solve *search* problems instead, i.e., when target function is constant. We show that Algorithm 1 models such ILP solvers. ILP solvers with other methods and Mixed Integer Linear Programming solvers use similar architectures and, thus, can be modelled similarly.

The search version of general branch-and-cut algorithm [8] is as follows:

1. Initialization: $S = \{\text{ILP}^0\}$ with ILP^0 the initial problem.
2. Termination: If $S = \emptyset$, return UNSAT.
3. Problem Select: Select and remove problem ILP^i from S.
4. Relaxation: Solve LP relaxation of ILP^i (as a search problem). If infeasible, go to step 2. Otherwise, if solution X^{iR} of LP relaxation is integral, return solution X^{iR}.
5. Add Cutting Planes: Add a cutting plane violating X^{iR} to relaxation and go to 4.
6. Partitioning: Find partition $\{C^{ij}\}_{j=1}^{j=k}$ of constraint set C^i of problem ILP^i. Create k subproblems ILP^{ij} for $j = 1, \cdots, k$, by restricting the feasible region of subproblem ILP^{ij} to C^{ij}. Add those k problems to S and go to step 2. Often, in practice, finding a partition is simplified by picking a variable x_i with non-integral value v_i in X^{iR} and returning partition $\{C^i \cup \{x_i \leq \lfloor v_i \rfloor\}, C^i \cup \{x_i \geq \lceil v_i \rceil\}\}$.

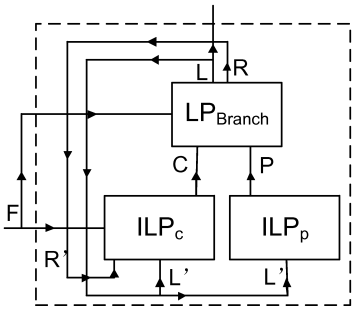


Fig. 3. Modular System Representing an ILP Solver

We use the modular system shown in figure 3 to represent the ILP solver. The ILP_c module takes the problem specification F, a set of assignments L' and a set of range information R' (which, in theory, describes assumptions about ranges of variables and, in practice, describes the branch information of an LP+Branch solver) as inputs and returns a set C. When all the assignments in L' are integral, C is empty, and if not, C represents a set of cutting planes conditioned by a subset of range information R', i.e., set of linear constraints that are violated by L' and all the set of assignments satisfy both F and R' also satisfy the set of cutting planes. The ILP_p module only takes the set of assignments L' as input and outputs a set of partitioning clauses P, such that when all the assignments in L' is integral, P is empty and when there is a non-integral assignment to a variable x, P is a set of partitioning clauses indicating that assignment to x should be either less than or equal to $\lfloor L'(x) \rfloor$ or greater than or equal to $\lceil L'(x) \rceil$. The LP_{Branch} module takes F, C and P as inputs and outputs the set of assignment L and the set of range information R such that L satisfies specification F, the range information R, the set of conditional cutting planes in C, and the set of partitioning clauses in P. We define the compound module ILP to be:

$$ILP := \pi_{\{F,L\}}(((ILP_c \cap ILP_p) \rhd LP_{Branch})[L = L'][R = R']).$$

The module ILP defined above is correct because all models satisfying it should have $C = B = \emptyset$ because, ILP_c contains structures in which, for every $(S, c) \in C$, which denotes cutting plane c under condition S, we have $S \subseteq R' = R$ and c is violated by $L' = L$. Furthermore, LP_{Branch} contains structures in which L satisfies both R and C, which indicates that either S is not the subset of R or L satisfies c. Thus C is empty. By a similar argument, one can prove that B also has to be empty.

We compute a model of this modular system by introducing a solver that interacts with all of the three modules above. We introduce an extended LP solver S which allows us to deal with disjunctions. S performs a depth-first-like search on disjunctive constraints, and runs its internal LP solver on non-disjunctive constraints plus the range information the search branch corresponds to. So, Partitioning action of ILP corresponds to adding a disjunctive constraint to the solver. All three modules above are associated with oracles O_c, O_p and O_{lp}, respectively. Exact constructions are similar to the ones in section 4.1. Here we only give a construction for the solver S:

Solver S is an extended LP solver, i.e., uses an internal LP solver. Let Br denote the set of branch constraints in S (constraints involving disjunctions) and L denote the set of pure linear constraints. When new constraint is added to S, S adds it to Br or L accordingly. S then performs a depth-first-like search on branch clauses, and, at branch i, passes $L \cup Br_i$ (a set of linear constraints specifying branch i) to its internal LP solver. Then, S returns $\langle SAT, \mathcal{B} \rangle$ if LP finds a model $\mathcal{B}$ in some branch, and $\langle UNSAT \rangle$ otherwise. Note that, by construction, S is complete and online.

Proposition 2. *1. Modular system ILP models the branch-and-cut-based ILP solver. 2. S is complete and online. 3. O_c, O_p and O_{lp} are CCAV oracles.*

There are many other solvers in ILP community that use some ILP or MILP solver as their low-level solver. It is not hard to observe that most of them also have similar architectures that can be closely mapped to our algorithm.

4.3 Modelling Constraint Answer Set Solvers

The Answer Set Programming (ASP) community puts a lot of effort into optimizing their solvers. One such effort addresses ASP programs with variables ranging over huge domains (for which, ASP solvers alone perform poorly due to the huge memory the grounding uses). However, embedding Constraint Programming (CP) techniques into ASP solving is proved useful because grounding such variables is partially avoided.

In [15], the authors extend the language of ASP and its reasoning method to avoid grounding of variables with large domains by using constraint solving techniques. The algorithm uses ASP and CP solvers as black boxes and non-deterministically extends a partial solution to the ASP part and checks it with the CP solver. Paper [16] presents another integration of answer set generation and constraint solving in which a traditional DPLL-like backtracking algorithm is used to embed the CP solver into the ASP solving.

Recently, the authors of [11] developed an improved hybrid solver which supports advanced backjumping and conflict-driven no good learning (CDNL) techniques. They show that their solver's performance is comparable to state-of-the-art SMT solvers. Paper [11] applies a partial grounding before running its algorithm, thus, it uses an algorithm on propositional level. A brief description of this algorithm follows: Starting from an empty set of assignments and nogoods, the algorithm gradually extends the partial assignments by both unit propagation in ASP and constraint propagation in CP. If a conflict occurs (during either unit propagation or constraint propagation), a nogood containing the corresponding unique implication point (UIP) is learnt and the algorithm backjumps to the decision level of the UIP. Otherwise, the algorithm decides on the truth value of one of the currently unassigned atoms and continues to apply the propagation. If the assignment becomes total, the CP oracle queries to check whether this is indeed a solution for the corresponding constraint satisfaction problem (CSP). This step is necessary because simply performing constraint propagation on the set of constraints, i.e., arc-consistency checking, is not sufficient to decide the feasibility of constraints.

The modular model of this solver is very similar to the one in Figure 2, except that we have module ASP instead of SAT and CP instead of ILP. The compound module $CASP$ is defined as:

$$CASP := \pi_{\{F,M,L\}}((CP \rhd ASP)[L = L']).$$

As a CDNL-like technique is also used in SMT solvers, the above algorithm is modelled similarly to Section 4.1. We define a solver S to be a CDNL-based ASP solver. We also define modules ASP and CP to deal with the ASP part and the CP part. They are both associated oracles similar to those described in Section 4.1. We do not include the details here as they are similar to the ones in section 4.1.

Note that one can add reasons and advices to an ASP solver safely in the form of conflict rules because stable model semantics is monotonic with respect to such rules. Also, practical CP solvers do not provide reasons for rejecting partial structures. This issue is dealt with in [11] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

5 Extension: Approximations

Almost all practical solvers use some kind of propagation technique. However, in a modular system, propagation is not possible in general because nothing is known in advance about a module. According to [6], it turns out that knowing only some general information about modules such as their totality and monotonicity or anti-monotonicity, one can hugely reduce the search space.

Moreover, paper [6] proposes two procedures to approximate models of what are informally called positive and negative feedbacks. These procedures correspond to least fixpoint and well-founded model computations (but in modular setting). Here, we extend Algorithm 1 using these procedures. The extended algorithm prunes the search space of a model by propagating information obtained by these approximation procedures to the solver. First, let us define some properties that a module may satisfy.

Definition 8 (Module Properties [6]). *Let M be a module and τ, τ' and τ'' be some subsets of M's vocabulary. M is said to be:*

1. **τ-total over a class C of structures** *if by restricting models of M to vocabulary τ we can obtain all structures in C.*
2. **τ-τ'-τ''-monotone (resp. τ-τ'-τ''-anti-monotone)** *if for all structures $\mathcal{B}$ and $\mathcal{B}'$ in M we have that if $\mathcal{B}|_\tau \sqsubseteq \mathcal{B}'|_\tau$ and $\mathcal{B}|_{\tau'} = \mathcal{B}'|_{\tau'}$ then $\mathcal{B}|_{\tau''} \sqsubseteq \mathcal{B}'|_{\tau''}$ (resp. $\mathcal{B}'|_{\tau''} \sqsubseteq \mathcal{B}|_{\tau''}$).*

In [6], it is shown that these properties are fairly general and that, given such properties about basic MX modules, one can derive similar properties about complex modules.

Now, we can restate the two approximation procedures from [6]. For $\mathcal{B}$ and $\mathcal{B}'$ over the same domain, but distinct vocabularies, let $\mathcal{B}||\mathcal{B}'$ denote the structure over that domain and $voc(\mathcal{B}) \cup voc(\mathcal{B}')$ where symbols in $voc(\mathcal{B})$ [resp. $voc(\mathcal{B}')$] are interpreted as in $\mathcal{B}$ [resp. $\mathcal{B}'$]. We first consider the case of a positive feedback, i.e., when relation R which increases monotonically when S increases is fed back into S itself. This procedure is defined for a module $M' := M[S = R]$ and partial structure $\mathcal{B}$ where M is $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-monotone and $\mathcal{B}$ gives total interpretation to τ. It defines a chain L_i of interpretations for S as follows:

$$\begin{array}{l} L_0 := S^{+^{\mathcal{B}}}, \\ L_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{L})} \text{ where } dom(\mathcal{L}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{L}} = L_i. \end{array} \tag{2}$$

The procedure for a negative feedback is similar, but for M being $\{S\}$-τ-$\{R\}$-anti-monotone. It defines an increasing sequence L_i and a decreasing sequence U_i which say what should be in added to S^+ and S^-. Here, n is the arity of relations R and S:

$$\begin{array}{c} L_0 := S^+, U_0 := [dom(\mathcal{A})]^n \backslash S^-, \\ L_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{U})} \text{ where } dom(\mathcal{U}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{U}} = U_i, \\ U_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{L})} \text{ where } dom(\mathcal{L}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{L}} = L_i. \end{array} \tag{3}$$

Now, Algorithm 1 can be extended to Algorithm 2 which uses the procedures above to find new propagated information which has to be true under the current assumptions, i.e., the current partial structure. This information is sent back to the solver to speed up the search process. Algorithm 2 needs a solver which can get propagated literals.

Data: Similar to Algorithm 1, but with modules' totality, monotonicity and anti-monotonicity properties given
Result: Structure $\mathcal{B}$ expands $\mathcal{A}$ and is in M
begin
  Initialize the solver S using the empty expansion of $\mathcal{A}$;
  while *true* **do**
    Let R be the state of S ;
    if $R = \langle UNSAT \rangle$ **then return** *Unsatisfiable* ;
    else if $R = \langle SAT, \mathcal{B} \rangle$ **then**
      Add the set of advices from oracles wrt $\mathcal{B}$ to S ;
      if M *does not accept* $\mathcal{B}$ **then**
        Find a module M_i in M such that M_i does not accept $\mathcal{B}|_{vocab(M_i)}$;
        Let ψ be the reason given by oracle O_i ;
        Add ψ to S ;
      else if $\mathcal{B}$ *is total* **then return** $\mathcal{B}$;
      else
        foreach *applicable positive feedback* $M_1 := M_2[T = T']$ **do**
          Let L^* be the limit of series L_i in Equation 2 ;
          Propagate $L^* \backslash {T^+}^{\mathcal{B}}$ to S ;
        foreach *applicable negative feedback* $M_1 := M_2[T = T']$ **do**
          Let $\langle L^*, U^* \rangle$ be the limit of series $\langle L_i, U_i \rangle$ in Equation 3 ;
          Propagate $(L^* \backslash {T^+}^{\mathcal{B}}) \cup \mathbf{not}([dom(\mathcal{A})]^n \backslash (U^* \cup {T^-}^{\mathcal{B}}))$ to S ;
end

Algorithm 2: Lazy Model Expansion with Approximation (Propagation)

6 Related Works

This paper is a continuation of [6] and proposes an algorithm for solving model expansion tasks in the modular setting. The modular framework of [6] expands the idea of model theoretic (and thus language independent) modelling of [13] and introduces the feedback operator and discusses some of the consequences (such as complexity implications) of this new operator. There are many other works on modularity in declarative programming that we only briefly review.

An early work on adding modularity to logic programs is [17]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. This is further generalized in [18] by considering the concept of modules in declarative programming and introducing modular equivalence in normal logic programs under the stable model semantics. This line of work is continued in [19] to define modularity for disjunctive logic programs. There are also other approaches to adding modularity to ASP languages and ID-Logic as described in [20–22].

The works mentioned earlier focus on the theory of modularity in declarative languages. However, there are also works that focus on the practice of modular declarative programming and, in particular, solving. These works generally fall into one of the two following categories:

The first category consists of practical modelling languages which incorporate other modelling languages. For example, X-ASP [23] and ASP-PROLOG [24] extend prolog with ASP. Also ESRA [25], ESSENCE [12] and Zinc [26] are CP languages extended with features from other languages. However, these approaches give priority to the host language while our modular setting gives equal weight to all modelling languages that are involved. It is important to note that, even in the presence of this distinction, such works have been very important in the development of this paper because they provide guidelines on how a practical solver deals with efficiency issues. We have emphasized on this point in Section 4.

The second category consists of the works done on multi-context systems. In [27], the authors introduce non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or explaining inconsistencies in non-monotonic multi-context systems [28–31]. However, these works do not consider the model expansion task. Moreover, the motivations of these works originate from distributed or partial knowledge, e.g., when agents interact or when trust or privacy issues are important. Despite these differences, the field of multi-context systems is very

relevant to our research. Investigating this connection as well as incorporating results from the research on multi-context system into our framework is our most important future research direction.

7 Conclusion

We addressed the problem of finding solutions to a modular system in the context of model expansion and proposed an algorithm which finds such solutions. We defined conditions on modules such that once satisfied, modules described with possibly different languages can participate in our algorithm. We argued that our algorithm captures the essence of practical solvers, by showing that DPLL(T) framework, ILP solvers and state-of-the-art combinations of ASP and CP are all specializations of our modular framework. We believe that our work bridges work done in different communities and contributes to cross-fertilization of the fields.

References

1. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI. (2005) 430–435
2. Kolokolova, A., Liu, Y., Mitchell, D., Ternovska, E.: On the complexity of model expansion. In: Proc., 17th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17), Springer (2010) 447–458 LNCS 6397.
3. Ternovska, E., Mitchell, D.: Declarative programming of search problems with built-in arithmetic. In: Proc. of IJCAI. (2009) 942–947
4. Tasharrofi, S., Ternovska, E.: Built-in arithmetic in knowledge representation languages. In: NonMon at 30 (Thirty Years of Nonmonotonic Reasoning). (October 2010)
5. Tasharrofi, S., Ternovska, E.: PBINT, a logic for modelling search problems involving arithmetic. In: Proc. 17th Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17), Springer (October 2010) LNCS 6397.
6. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: 8th International Symposium Frontiers of Combining Systems (FroCoS). (October 2011)
7. Niemelä, I.: Integrating answer set programming and satisfiability modulo theories. In: LPNMR. Volume 5753 of LNCS., Springer-Verlag (2009) 3–3
8. Pardalos, P., Resende, M.: Handbook of applied optimization. Volume 126. Oxford University Press New York; (2002)
9. Sebastiani, R.: Lazy Satisfiability Modulo Theories. JSAT **3** (2007) 141–224
10. De Cat, B., Denecker, M.: DPLL(Agg): An efficient SMT module for aggregates. In: LaSh'10 Workshop. (2010)
11. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proc. of ICLP'09. LNCS, Springer-Verlag (2009) 235–249
12. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13** (2008) 268–306
13. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Proc. of LPNMR. Volume 5753 of LNCS., Springer-Verlag (2009) 155–168
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (November 2006) 937–977
15. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. (2005) 52–66
16. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53** (2008) 251–287
17. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Proc. of LPNMR, Springer-Verlag (1997) 290–309
18. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: the Proc. of NMR'06. (2006) 10–18
19. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Proc. of LPNMR. Volume 4483 of LNAI. (2007) 175–187
20. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: ICLP'06. LNCS. (2006) 376–390
21. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: SEA. (2007) 41–55

22. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. Transactions on Computational Logic **9**(2) (2008) 1–51
23. Swift, T., Warren, D.S.: The XSB System. (2009)
24. Elkhatib, O., Pontelli, E., Son, T.: ASP- PROLOG: A System for Reasoning about Answer Set Programs in Prolog. In: the Proc. of PADL'04. (2004) 148–162
25. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. Logic Based Program Synthesis and Transformation (2004) 214–232
26. de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. Principles and Practice of Constraint Programming-CP 2006 700–705
27. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of the 22nd national conference on Artificial intelligence - Volume 1, AAAI Press (2007) 385–390
28. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The dmcs solver for distributed nonmonotonic multi-context systems. In Janhunen, T., Niemelä, I., eds.: Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings. Volume 6341 of Lecture Notes in Computer Science., Springer (2010) 352–355
29. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The mcs-ie system for explaining inconsistency in multi-context systems. In Janhunen, T., Niemelä, I., eds.: Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings. Volume 6341 of Lecture Notes in Computer Science., Springer (2010) 356–359
30. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In Lin, F., Sattler, U., Truszczynski, M., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010, AAAI Press (2010)
31. Eiter, T., Fink, M., Schüller, P.: Approximations for explanations of inconsistency in partially known multi-context systems. In Delgrande, J.P., Faber, W., eds.: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Volume 6645 of Lecture Notes in Computer Science., Springer (2011) 107–119

FdConfig: A Constraint-Based Interactive Product Configurator

Denny Schneeweiss and Petra Hofstedt

Brandenburg University of Technology, Cottbus
schneden@tu-cottbus.de and hofstedt@informatik.tu-cottbus.de

Abstract. We present a constraint-based approach to interactive product configuration. Our configurator tool *FdConfig* is based on feature models for the representation of the product domain. Such models can be directly mapped into constraint satisfaction problems and dealt with by appropriate constraint solvers. During the interactive configuration process the user generates new constraints as a result of his configuration decisions and even may retract constraints posted earlier. We discuss the configuration process, explain the underlying techniques and show optimizations.

1 Introduction

Product lines for mass customization [22] allow to fulfill the needs and requirements of the individual consumer while keeping the production costs low. They enhance extensibility and maintainance by reusing the common core of the set of all products.

Product configuration describes the process of specifying a product according to user-specific needs based on the description of all possible (valid) products (the search space). When done *interactively*, the user specifies the features of the product step-by-step according to his requirements, thus, gradually shrinking the search space of the configuration problem. This interactive configuration process is supported by a software tool, the *configurator*.

In this paper, we present an approach on interactive product configuration based on constraint programming techniques. Building on constraints enables us to equip our interactive product configurator *FdConfig* with functionality and expressiveness exceeding traditional approaches but at the cost of performance penalty which must be dealt with in turn.

The paper is structured as follows: In Sect. 2 we briefly review the area of interactive configuration methods and discuss related work. Section 3 introduces important notions from the constraint paradigm as needed for the discussion of our approach. We present the constraint-based interactive product configurator *FdConfig* in Sect. 4. There, we introduce *FdFeatures*, a language for the definition of feature models, it's transformation into constraint problems, and the configuration process using *FdConfig*. Furthermore, we discuss optimizations and improvements by analyses and multithreading. Section 5 draws a conclusion and points out directions of future research.

2 Interactive Configuration Methods

An interactive product configurator is a tool which allows the user to specify a product according to his specific needs based on the common core of the set of all products of a product line. This process can be done interactively, i.e. in a step-wise fashion, thus gradually shrinking the search space of the configuration problem.

For the sake of applicability and user-friendliness, a configurator requires a number of properties like backtrack-freeness, completeness, order-independent retraction of decisions, short response times and others. These strongly depend on the method[1] underlying the configurator system. Cost optimization and arithmetic constraints are a desired functionality too, but these are seldom supported or only provided in a very restricted form.

[1] For a discussion of the solutions methods see below.

While *completeness* ensures that no solutions are lost, *backtrack-freeness* [7, 20] guarantees that the configurator only offers decision alternatives for which solutions remain. Thus, the user can always generate a valid solution from the current configuration state and does not need to unwind a decision (i.e. he does not need to backtrack). The *Calculate Valid Domains (CVD) function* [7] of a configurator realizes this latter property.

Feature models are particularly used in the context of software product line engineering to support the reuse when building software-intensive products. However, they are of course applicable to many other product line domains. They stem from the *feature oriented domain analysis methodology* (FODA) [14].

A feature model describes a product domain by a combination of features, i.e. specific aspects of the product which the user can configure by instantiation and further constraints. A product line is given by the set of possible combinations of feature alternatives.

The semantics of feature models is typically mapped to propositional logics [11] and can accordingly be mapped onto a restricted class of constraint satisfaction problems (cf. Sect. 3), namely constraints of the Boolean domain. While many approaches in the literature (e.g. [7, 8, 12]) only consider constraints of the Boolean domain (including equality constraints), Benavides et.al. [1] discuss the realization of arithmetic computations and cost optimization in a feature model (by so-called "extra-functional features") which can be represented by general constraint problems.

Solution techniques applied to the interactive configuration problem have been compared by Hadzic et.al. [7, 8] and Benavides et.al. [2]. They mainly distinguish approaches based on propositional logic on the one hand and on constraint programming on the other hand.

When using *propositional logic* based approaches, configuration problems are restricted to logic connectives and equality constraints (see e.g. [7, 21]). Arithmetic expressions are excluded because of the underlying solution methods. These approaches perform in two steps. First, the feature model is translated into a propositional formula. In the second step the formula is solved (satisfiability checking, computation of solutions) by appropriate solvers, in particular *SAT solvers* (as in [12]) and *BDD-based solvers* (see e.g. [8, 20]). BDD-based solvers translate the propositional formula into a compact representation, the BDD (binary decision diagram). While many operations on BDDs can be implemented efficiently, the structure of the BDD is crucial as a bad variable ordering may result in exponential size and, thus, in memory blow up. Therefore the compilation of the BDD is done in an offline phase, so a suitable variable ordering can be found and the BDD's size becomes reasonably small.

Feature models can be naturally mapped into *constraint systems*, in particular into CSPs. There are some approaches [1, 21] using this correspondence to deal with interactive configuration problems. These typically work as follows: The feature model is translated into a constraint satisfaction problem (CSP, see Definition 1 below) and afterwards analysed by a CSP solver. Using this approach, no pre-compilation is necessary. In general it is possible to use predicate logic expressions and arithmetic in the feature definitions, even if this is not realized in the above mentioned approaches.

Transformations of feature models into programs of CLP languages (i.e. Prolog systems with constraints) have been shown recently in [15, 17]. However, beside the transformation target beeing different from ours, these approaches do not focus on using these methods for interactive configuration.

Since our *FdConfig* tool aims primarily at the software engineering community as the main users of feature models, we decided in favour of a JAVA-implementation, which would make later integration with common software development infrastructure like *Eclipse* more easy.

Benavides et.al. [2] elaborately compare the approaches sketched above, particularly with respect to *performance* and *expressiveness or supported operations, resp.* They point out that CSP-based approaches, in contrast to others, can allow extra functional features [1, 14] and, in addition, arithmetic and optimization. Furthermore, they state that "the available results suggest" that constraint-based and propositional logic-based approaches "provide similar performance", except for the BDD-approach which "seems to be an exception as it provides much faster execution times", but with the major drawback of BDDs having worst-case exponential size.

Extended feature models with numerical attributes, arithmetic, and optimization are denominated as an important challenge in interactive configuration by Benavides et.al. [2]. Our approach aims at this challenge. The main idea is to follow the constraint-based approach while using the combination of different

constraint methods and concurrency to deal with the computational cost. At this, a major challange is to support the user when making and withdrawing decisions in an interactive process.

3 Constraint Programming

Feature models can directly be mapped into corresponding constraint problems. We will discuss this approach more detailed in Sect. 4.1 but introduce the necessary notions from the constraint paradigm here.

Constraints are predicate logic formulae which express relations between the elements or objects of the problem. They are classified into *constraint domains* (see [9, 18]), e.g. linear constraints, Boolean constraints and finite domain constraints. This partitioning is due to the different applicable constraint solution algorithms implemented in so-called constraint solvers (see below).

Considering feature models as constraint problems, the domains of the involved variables are a priori finite.[2] Thus, we consider a particular class of constraints: *finite domain constraints.* Finite domain constraint problems are given by means of constraint satisfaction problems.

Definition 1 (CSP). *A* Constraint Satisfaction Problem (CSP) *is a triple* $P = (X, D, C)$*, where* $X = \{x_1, \ldots, x_n\}$ *is a finite set of variables,* $D = (D_1, \ldots, D_n)$ *is an* n*-tuple of their respective finite domains, and* C *is a conjunction of constraints over* X*.*

Definition 2 (solution). *A* solution *of a CSP* P *is a valuation* $\varsigma : X \rightarrow \bigcup_{i \in \{1,\ldots,n\}} D_i$ *with* $\varsigma(x_i) \in D_i$ *which satisfies the constraint conjunction* C*.*

A CSP can have one solution or a number of solutions, or it can be unsatisfiable. Optimization functions may also be given which specify optimal solutions.

Example 1. Consider a CSP $P = (X, D, C)$ with the set $X = \{Cost, Color, Band\}$ of variables and their respective domains $D = (D_{Cost}, D_{Color}, D_{Band})$ with $D_{Color} = \{Red, Gold, Black, Blue\}$, $D_{Cost} = \{0, ..., 1400\}$, and $D_{Band} = \{700, 800, 1000\}$.

$C = (Band = 700 \rightarrow Color = Blue) \wedge (Cost = Band + 500)$ is a conjunction of constraints over the variables from X.

Solutions of the CSP P are e.g. ς_1 with $\varsigma_1(Cost) = 1200$, $\varsigma_1(Color) = Blue$, and $\varsigma_1(Band) = 700$ which is also denoted by $\varsigma_1 = \{Cost/1200, Color/Blue, Band/700\}$ and $\varsigma_2 = \{Cost/1300, Color/Red, Band/800\}$.

Constraint solvers are sets or libraries of tests and operations on constraints, which are able to check the satisfiability of constraints and to compute solutions and implications of constraints.

CSPs are typically solved by narrowing the variable's domains using search nested with consistency techniques (e.g. node, arc, and path consistency). Given a CSP, in the first step consistency techniques are applied. Such consistency checking algorithms work on n-ary constraints and try to remove values from the variables domains which cannot be elements of solutions. Afterwards, search is initiated, e.g. using backtracking, where we assign domain values to variables and perform consistency techniques to narrow the other variable's domains again. This search process is controlled by heuristics on variable and value ordering (for the complete process, see [18]).

There are some finite domain constraint solver libraries available, for example the JAVA-libraries CHOCO [3] as well as JACOP [10] and the C++-library GECODE [6]. We decided in favor of the freely available CHOCO library which is under continuous development.

Additionally, we need the notions of global consistency and of valid domains.

Definition 3 (global consistency, see [18]). *A CSP is* i*-consistent iff given any consistent instantiation of* $i - 1$ *variables, there exists an instantiation of any* i*th variable such that the* i *values taken together satisfy all of the constraints among the* i *variables. A CSP* $P = (X, D, C)$ *is globally consistent, if it is* i*-consistent for every* i*,* $1 \leq i \leq n$*, where* n *is the number of variables of* C*.*

[2] An extension to infinite domains would be possible, in general.

Definition 4 (valid domains). *Given a CSP P, the* valid domains *of P is an n-tuple $D_{vd} = (D_{vd,1}, \ldots, D_{vd,n})$ such that each $D_{vd,i} \subseteq D_i$ contains exactly the values which are elements of solutions of P.*

So, if a CSP is gobally consistent, then its domains are valid domains.

Example 2. (continuation of Example 1) The valid domains of the CSP P is $D_{vd} = (\{1200, 1300\}, D_{Color}, \{700, 800\})$.

4 The Interactive Configurator *FdConfig*

Our approach on interactive configuration consists of two phases: In the first phase a feature model is analysed and then transformed into a CSP and passed to the CHOCO solver. Afterwards the interactive configuration phase follows.

Figure 1 illustrates the analysis and transformation phase. *FdConfig* uses *FdFeatures* files as input. *FdFeatures* is a textual domain specific language for extended feature models which supports integer feature attributes and arithmetic constraints. An *FdFeatures* parser reads the input-file and creates the feature model which is transformed into a CHOCO CSP. Section 4.1 describes the language *FdFeatures* and the transformations in greater detail. Additionally, a quick pre-calculation of the variable's domains is performed. It generates redundant constraints which, nevertheless, help to improve the solver's performace.[3] This *domain analysis* is covered in Sect. 4.2.

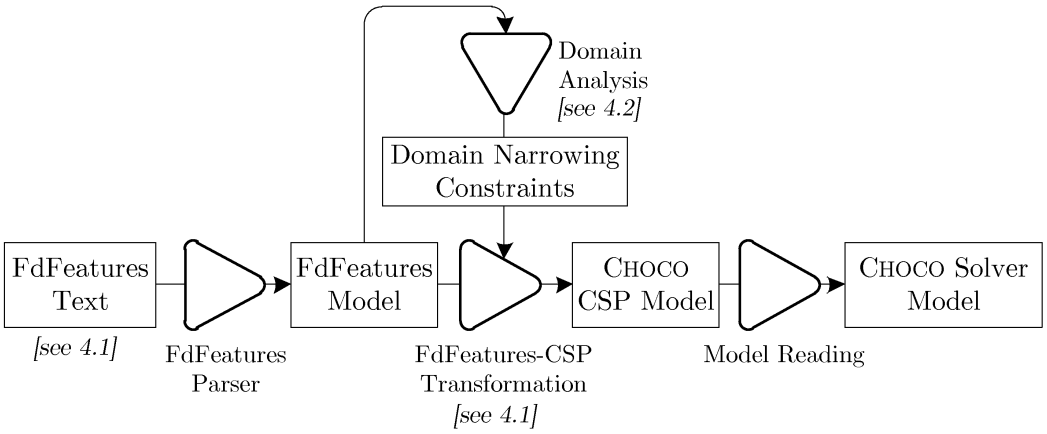


Fig. 1. Transformations performed before the user can start configuring

In the second phase, the generated CSP is passed to the CHOCO solver which reads the model and creates an internal representation from it: the solver model. Then the solver is started to perform an initial calculation of consequence decisions that yield from the constraints in the *FdFeatures* model. Afterwards, the user can start with the interactive configuration. The implementation of this process is explained in Sect. 4.3. Section 4.4 describes the reduction of response times by using multithreading.

[3] In constraint programming, the generation of redundant constraints from a given constraint problem is a frequently used method which helps to speed up the solver (see [18], Sect. 12.4.5). Note that the elimination of verification-irrelevant features and constraints (i.e. "redundant relationships", [24]) from feature models with the aim of reducing the problem size is a different concept.

4.1 *FdFeatures* Models and CSPs

FdConfig provides *FdFeatures* as a language for the definition of feature models based on the approach of [5]. *FdFeatures* borrows from the TEXTUAL VARIABILITY LANGUAGE (TVL, [4]) but was adapted for our needs (e.g. including support for the realization of the user interface, certain detailed language elements and syntactic sugar). *FdFeatures* has been implemented using XTEXT [23].

An *FdFeatures feature model* in general has a tree structure, i.e. there is a distinguished root feature which stands for the item to be configured, but apart from this behaves like any other feature. The model may have additional constraints between (sub-)features and their attributes which, in fact, makes the tree a general graph. Nevertheless, the tree structure is dominant.

A feature may consist of sub-features and attributes (both in general optional), where, following the approach of [5], the sub-features can be organized in *feature groups*. A feature group allows to describe whether one, some, or all of the sub-features must be included in the configured product.

With similar effects, *features* can be specified to be **mandatory** or **optional**. Furthermore, features may exclude or require other features.

Example 3. Consider the cut-out of a feature model of events organized by an event agency in Listing 4.1.[4] For an event (the root feature) we can optionally order a band and a stage, but we must order a carpet (e.g. for a film premiere or a wedding) and colored balloons. These are all modeled as sub-features (which are not organized in a feature group here). Ordering a band makes a stage necessary, expressed by the **requires**-statement in Line 11.

FdFeatures supports three kinds of feature *attributes:* integers, enumerations, and Boolean values.

Example 4. (continuation of Example 3) The feature *Carpet* is determined by several attributes, including an enumeration attribute *Color*, whose domain elements must be given explicitly and a Boolean attribute *SlipResistance* .[5]

Length and *Breadth* are integer attributes. While *Breadth* is specified by an interval, *Length* is unbounded. As we can see by the attributes of *ColoredBalloons*, the domain of an integer attribute can also be specified by a finite set (*Amount*, Line 20) or even by an arithmetic formula (*Cost*, Line 25). The definition of Boolean attributes is also possible using Boolean expressions (but is left out in our example).

The domain definition of *PriceReduction*.*Coupon* (Lines 22, 23) uses *Guards* to define the attribute domain depending on the configuration state (**ifIn** and **ifOut** correspond to selected and deleted, resp.) of the feature (here *PriceReduction*). Furthermore, it is possible to define *constraints* on attributes and features, also accessing the configuration state of a feature as shown in Line 15. This constraint makes sure that the Blues-band plays in an adequate ambiance.

The *transformation* into a CHOCO CSP is straightforward, for details see [19]. In general, our transformation is similar to these of [15, 17]. Differences come from the fact that the transformation target of these approaches are CLP languages and they aim at feature model analysis in contrast to interactive configuration, as does *FdConfig*. We show an example of the generated CSP in a mathematical notation and leave out the CHOCO constraint syntax for reasons of space limitations.

Example 5. The following CSP is generated from Listing 4.1 (where $CBal$ stands for $ColoredBalloons$, $PRed$ for $PriceReduction$, and $SRes$ for $SlipResistance$ resp.). Note that we do not enumerate the set of variables X explicitely and give the domains D by means of element constraints $C_{Domains}$.

$$CSP = C_{Domains} \wedge C \text{ with}$$

$$C_{Domains} = Event, Carpet, Band, Stage, CBal \in \{False, True\} \wedge$$
$$CBal.PRed, Carpet.SRes \in \{False, True\} \wedge$$

[4] The description of certain features and attributes, which are not necessary for the understanding of this example and the concepts behind, is left out and represented by "..." in the program.

[5] Note that the domain values of an enumeration may be assigned to integer values as e.g. done for the attribute *Band.Type* in Line 12.

Listing 4.1 Feature model of an event organized by an event agency (cut-out)

```
1  root feature Event {
2    enum Discount in {Gold = 8, Staff = 3, None = 0};
3    feature Carpet {
4      int Length;
5      int Breadth in [50..300];
6      enum Color in {Red, Gold, Black, Blue};
7      bool SlipResistance;
8      int Cost is ...
9    }
10   feature Band : optional {
11     requires Stage;
12     enum Type in {Classic = 1000, Blues=700, Rock=900};
13   }
14   feature Stage : optional { ... }
15   constraint BluesOnBlueCarpet
16     Band is selected and Band.Type = Blues ->
17       Carpet.Color = Blue
18   }
19   feature ColoredBalloons {
20     int Amount in {500, 1000, 2500, 5000, 10000};
21     feature PriceReduction {
22       int Coupon ifIn: is 1000
23                  ifOut: is 0;
24     }
25     int Cost is Amount * 3 - PriceReduction.Coupon;
26   }
27   int OverallCost is Carpet.Cost + Band.Type + ... +
28       (ColoredBalloons.Cost + ...) * (100-Discount)/100;
29 }
```

$$Discount \in \{0,3,8\} \wedge Band.Type \in \{700,900,1000\} \wedge$$
$$Carpet.Length \in [-2^{31}, 2^{31}-1] \wedge Carpet.Breath \in [50,300] \wedge$$
$$Carpet.Color \in [0,3] \wedge Carpet.Cost = ... \wedge$$
$$CBal.Amount \in \{500,1000,2500,5000,10000\} \wedge$$
$$CBal.Cost \in [-2^{31}, 2^{31}-1] \wedge$$
$$CBal.PRed.Coupon \in [-2^{31}, 2^{31}-1] \wedge$$
$$OverallCost \in [-2^{31}, 2^{31}-1] \text{ and}$$

$$C = (Carpet \vee Band \vee Stage \vee CBal \rightarrow Event) \wedge$$
$$(CBal.PRed \rightarrow CBal) \wedge (Band \rightarrow Stage) \wedge$$
$$((Band \wedge Band.Type = 700) \rightarrow Carpet.Color = 3) \wedge$$
$$(CBal.PRed \rightarrow CBal.PRed.Coupon = 1000) \wedge$$
$$(\neg\, CBal.PRed \rightarrow CBal.PRed.Coupon = 0) \wedge$$
$$(CBal.Cost = CBal.Amount * 3 - CBal.PRed.Coupon) \wedge \ldots$$

4.2 Domain Analysis

In *FdFeatures* the specification of an attribute's base domain is optional. If no domain is given by the user, as e.g. for *Carpet*.*Length* or *ColoredBalloons*.*Cost* in Listing 4.1, it is set by default to the maximal possible domain of the corresponding attribute type. For example, for integer attributes the maximal domain is $[-2^{31}, 2^{31}-1]$ which we denote by $MAXDOM$ in the following.

When the CHOCO solver computes the valid domains of the CSP in the second phase of our approach (cf. Sect. 4.3), this may become time consuming. The solver must establish global consistency. Thus, up to $4.3 * 10^9$ values must be checked for every attribute (or its corresponding variable, resp.) with $MAXDOM$. Of course, we cannot require the user to specify attribute domains just big enough to contain all solutions, in particular, because a manual estimation of the base domain can be very difficult for complex feature models. Thus, we apply an automatic pre-analysis to the feature model which is merged with the CSP generated from the model.

Our *domain analysis* aims at an approximate yet quick pre-calculation of the base domains of variables using knowledge about the feature model's structure. We only consider integer attributes, as enumerated attributes will in general have small domains. The analysis is based on interval arithmetics [16] which allow a fast approximation of the variable's minimum and maximum values by calculating with intervals instead of single domain values.

The domain DOM_{FM} of an attribute in *FdFeatures* can be specified directly by giving a single value or a set or interval, resp. of values. Additionally, it is possible to specify particular sub-domains depending on the configuration state, i.e. IN_{FM} and OUT_{FM} in case the attribute is selected or deleted, resp. Furthermore, arithmetic expressions can be used to specify the domain or sub-domains. We determine DOM_{FM}, IN_{FM}, and OUT_{FM} in form of intervals from the attribute expressions, where enumarations are handled as intervals, too.

Starting from these domains, we calculate the narrowed base domain $BASEDOM$, and new sub-domains IN and OUT as follows (where we take arithmetic expressions into consideration):

$$BASEDOM = (IN_{FM} \cup OUT_{FM}) \cap DOM_{FM} \quad (1)$$
$$IN = BASEDOM \cap IN_{FM} \quad (2)$$
$$OUT = BASEDOM \cap OUT_{FM} \quad (3)$$

The intervals for the incorporated arithmetic expressions are determined by traversing their formula tree. The leafs are either elementary expressions or references to other attributes, in case of which the domain of the referenced attribute must be calculated first. The analysis of cyclic formulae is interrupted and $MAXDOM$ is used instead, leaving domain narrowing to the CHOCO solver, which uses accurate but time consuming consistency techniques.

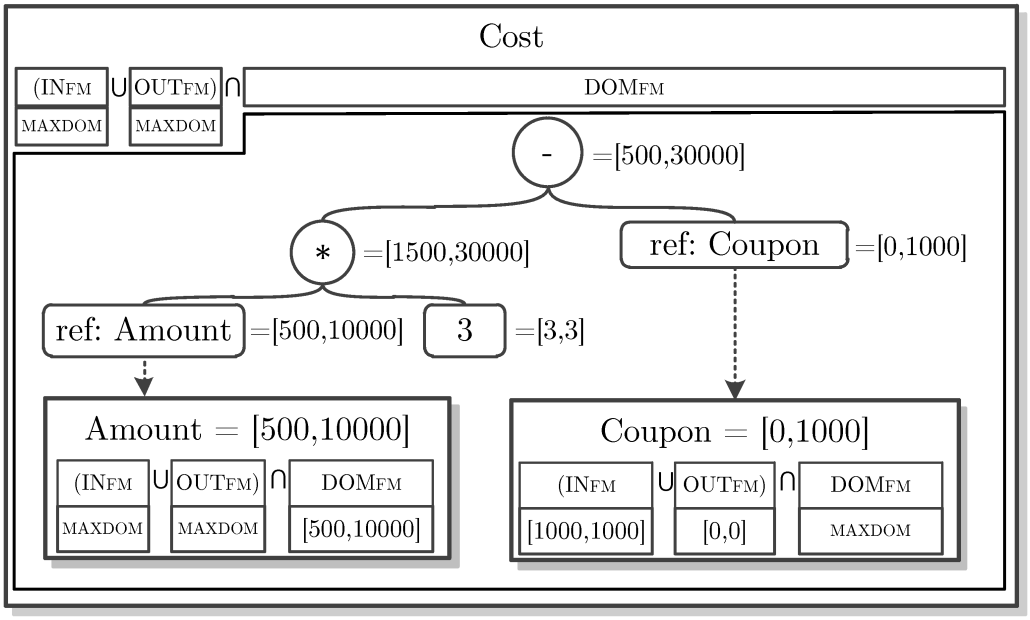


Fig. 2. Domain analysis of $BASEDOM$ of the attribute *ColoredBalloons.Cost*

Example 6. Consider the pre-calculation of the base domain $BASEDOM$ of the attribute *ColoredBalloons.Cost* (Line 25 of Listing 4.1). Figure 2 illustrates the calculation.

For the attribute under consideration, only the set DOM_{FM} is specified by means of an arithmetic expression, while IN_{FM} and OUT_{FM} both default to $MAXDOM$. During the analysis, the formula tree of the arithmetic expression is traversed. Dashed arrows depict the domain analysis of a referenced attribute which is shown in it's own box.

In the beginning the analysis moves to the first leaf: a reference to the attribute *Amount*. The determination of the base sets is trivial as only DOM_{FM} is defined as an enumeration of integers which yields an interval $[500, 10000]$ using Equation 1. The right operand of the multiplication is a constant value, which is turned into the point interval $[3, 3]$ resulting in the intermediate result $[500, 10000] * [3, 3] = [1500, 30000]$. The analysis of the attribute *Coupon* yields $BASEDOM = [0, 1000]$ from $IN_{FM} = [1000, 1000]$, $OUT_{FM} = [0, 0]$ and $DOM_{FM} = MAXDOM$ (again using Equation 1). Finally, the analysis returns to the root attribute *ColoredBalloons.Cost* and performs the subtraction with result $BASEDOM = [1500, 30000] - [0, 1000] = [500, 30000]$.

From the $BASEDOM$ intervals of the attributes the respective sub-domains IN and OUT can be inferred by means of the Equations 2 and 3 (not shown in the figure).

Example 7. (continuation of Examples 5 and 6) The domain analysis yields the following domain constraints as an update on the generated CSP of our event feature model.

$$C'_{Domains} = \ldots \wedge \\ CBal.Cost \in [500, 30000] \wedge \\ CBal.PRed.Coupon \in \{0, 1000\} \wedge \ldots$$

Note, that for computed intervals we finally build intersections in case the domain was initially given by enumerations or single values. This yields the two-element set for $CBal.PRed.Coupon$.

4.3 The Configuration Phase

The second phase of our approach, i.e. the configuration phase, starts with the initialization of *FdConfig* before the user can start with the interactive configuration process.

Model pre-processing. The solver reads the CSP-model and performs a feasibility check (e.g. by finding the first solution). If successful, the configurator computes the valid domains as initial *model consequences* that derive from the CSP. The calculation of these model consequences is performed in the same way as the user consequences are calculated later on in the interactive user configuration phase (see below). However, once the model consequences have been computed, they are immutable during the interactive configuration as they don't depend on the user decisions.

The current, global consistent state of the solver is recorded. To this *ground level state* the solver can be reset when, after a retraction of user constraints, a re-computation of the valid domains becomes necessary.

User configuration. The user starts a configuration step by executing a configuration action. This is either a configuration decision, i.e. limiting the domain of a feature- or attribute variable which manifests as a user constraint or the retraction of a decision made earlier. In this case the corresponding user constraint is removed from the constraint system. User decisions are posted by *FdConfig* to the solver as user constraints.

Now, the solver is activated to establish global consistency and to find all solutions of the constraint system. These are evaluated to derive the valid domains. Since the valid domains define the configuration options available to the user in the next configuration step, the constraint system always remains feasible after a user decision.

After the user consequences have been computed, the user interface is updated accordingly and the user can perform the next configuration action.

In the usual modus operandi for FD solvers, a CSP is once declared and then read by the solver which computes and returns solutions. In contrast, for interactive configuration we need to re-calculate sets of solutions again and again because a sub-set of the constraints (the user constraints) keeps changing over time as a result of the user making configuration decisions.

As the solver maintains a heavyweight internal representation of the constraint system and reading the CSP-model as well as establishing consistency are time consuming, the option of re-creating the solver for every user decision is inapplicable. Therefore we control the solver from outside by utilizing its backtracking infrastructure and reset the solver into the aformentioned ground level state in case a user decision has been retracted.

4.4 Improving the User Experience by Multithreading

When computing the valid domains of the variables, the constraint solver must establish global consistency, and thus, potentially find all solutions of the CSP. This calculation may be time consuming depending on the size and complexity of the feature model (and the CSP it was transformed into, resp.). Furthermore, the GUI would not be updated or process user input during this calculation. The program would appear to be frozen.

Therefore we introduced multithreading with the solver running in a background thread, thus, allowing the GUI to be updated and accept user input during a long running computation. However, as the user would still have to wait for the calculation to complete before he can enter another configuration decision, the multithreading structure has been extended as follows:

The elements of the valid domains are collected gradually with the computation of the set of solutions still in progress. Whenever new elements have been found, they are immediately displayed in the GUI and made available for configuration decisions. Elements, that did not yet occur in a solution, are greyed out and disabled for user decisions. If the user makes a decision, the background calculation is interrupted and restarted with the changed set of user decisions.

In the sequential model the valid domains were calculated in one go and then evaluated to generate consequence decisions if necessary. If, for example, the valid domain of a feature A was found to be $D_{vd,A} = \{true\}$ this resulted in a consequence decision forcing the feature to be *selected*[6].

[6] Likewise $D_{vd,A} = \{false\}$ results in a *removed* feature and $D_{vd} = \{true, false\}$ in the *undecided* state, where the user can decide.

With multithreading we have to re-evaluate the valid domains whenever new elements are found during the calculation process. This results in changing consequence decisions while the computation has not finished. For example, the valid domain of feature A can become $D_{vd,A} = \{true\}$ during the computation process at first, creating the consequence decision that A must be selected. However, as the result of new solutions the valid domain might later become $D_{vd,A} = \{true, false\}$, thus making the consequence decision disappear again. Attributes are handled similarly, as single value domains (interpreted as consequence decisions to select this particular value) may become multi-value domains later on. The GUI flags these consequence decisions as *incomplete*, so the user can see that further configuration options might become available. On the completion of the computation process, this flag is removed.

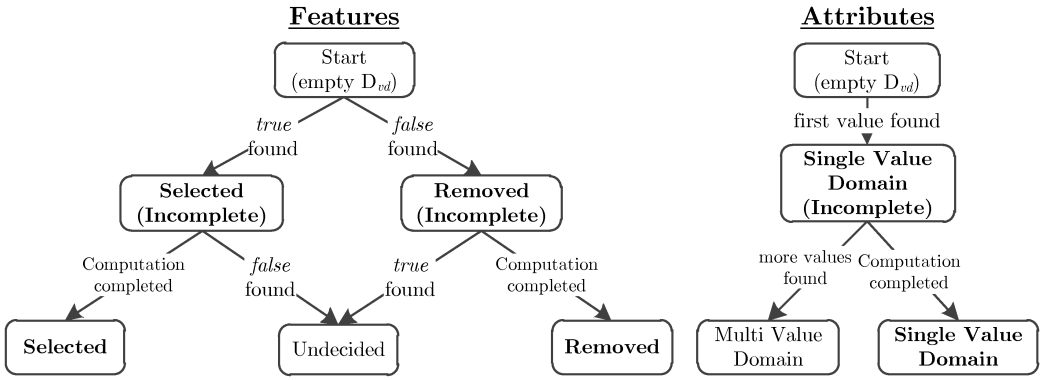


Fig. 3. Intermediate states of valid domains for features and attributes with multithreading

Figure 3 illustrates the different states for feature and attribute domains, resulting from the multithreading approach. Consequence decisions are drawn in bold typeface. Furthermore Fig. 4 shows a screenshot of the *FdConfig* tool during a long running calculation of the valid domains. Incomplete consequence decisions are visible, i.e. for the attribute *ColoredBalloons.Cost*, whose valid domain has exactly one element (14990) at the moment. The other elements were either eliminated by the user or have not yet occurred in a solution (displayed in grey).

First experiments show that this multithreading approach leads to a smoother, more fluent user experience when performing product configuration. Since reaching the goal of calculating the valid domains in under 250 msec[7] is currently not realistic with the underlying solvers, this enhancement is a good compromise as configuration options will become available very quickly.

5 Conclusion and Future Work

In this paper we discuss an approach on interactive product configuration based on constraint techniques, which was implemented in our configurator tool *FdConfig*. We gave an overview of the product configuration domain, feature models, and constraint programming in this context and introduced our approach.

In *FdConfig* we employ a finite domain constraint solver that enables us to deal with integer attributes and arithmetic constraints in extended feature models. These constraints are usually not supported in traditional approaches (e.g. SAT, BDDs) or only in restricted forms. However, this enhanced expressiveness comes at the cost of performance penalties. We deal with this by applying a preliminary domain analysis in order to relief the solver of unnecessary computation time for establishing consistency. Furthermore, we

[7] A response time of about this duration is considered desirable, as this still gives the user the impression to work in real time [7].

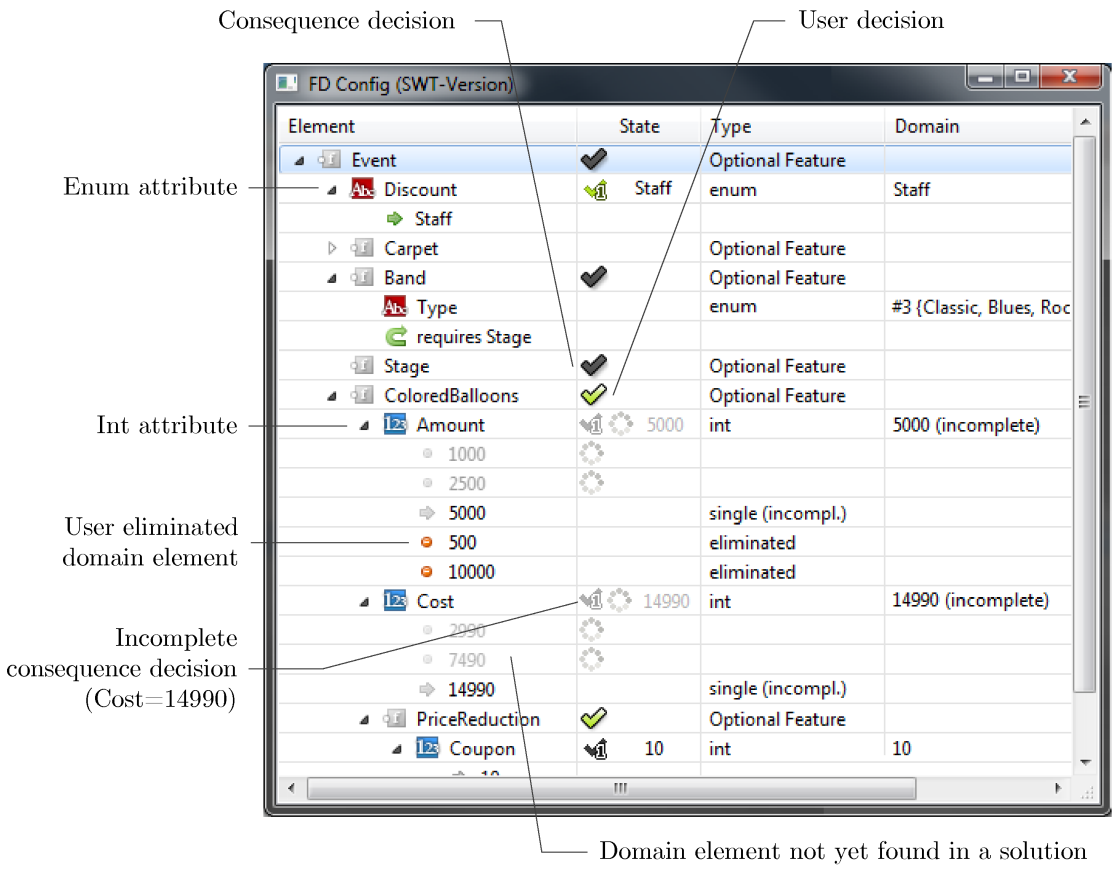


Fig. 4. Screenshot of *FdConfig* during a CVD calculation

use a multithreading approach to enhance the user experience. This allows the user to continue configuring in a limited way, even if the overall computation has not yet finished.

Future work will include the further development of the multithreading approach. We plan to incorporate multiple solvers that might use different computation strategies. For example, the feature model element with the current GUI focus could be taken into account. This focus-based computation strategy could additionally improve user friendliness: Domain elements, that the user might want to configure most likely, would become available more quickly for configuration decisions.

Also a more subtle handling of the non-chronological retraction of constraints promises improvement but needs further investigation.

In order to improve the overall performance we consider adding support for compilation-based approaches (i.e. BDDs). These could be integrated with the solver in the form of custom constraints to speed up the search. If a pre-compiled version of a feature model is available, the implementation of these constraints could access the BDD. Otherwise the regular solution methods would be applied.

Transformation-based optimizations should be investigated, too. E.g. [13] use a clustering optimization to reduce the number of constraint-variables and constraints. Using feature models ([13] directly use constraints) may support or even inherently realize a form of clustering.

Another optimization is presented in [17]. The authors discuss the improvement of efficiency when solving CSPs as transformation results due to a reformulation of particular boolean constraints into arithmetic constraints. While this representation is available in our approach too, the examination of similar optimizations may be worth considering in the future. In the approach of [17] the structure of feature models is not preserved. This holds optimization potential as well, but must be done sensitive to retain a mapping to the feature model to allow an interactive configuration process as needed in our approach.

References

1. David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In *International Conference Advanced Information Systems Engineering (CAISE)*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.
2. David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
3. ChocoSolver. http://www.emn.fr/z-info/choco-solver/. last visited 2011-07-07.
4. Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.
5. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
6. Gecode – Generic Constraint Development Environment. http://www.gecode.org. last visited 2011-07-07.
7. Tarik Hadzic and Henrik Reif Andersen. An introduction to solving interactive configuration problems. Technical Report TR-2004-49, The IT University of Copenhagen, 2004.
8. Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems (PETO)*, pages 131–138, 2004.
9. Petra Hofstedt. *Multiparadigm Constraint Programming Languages*. Springer, 2011.
10. JaCoP – Java Constraint Programming solver. http://jacop.osolpro.com/. last visited 2011-07-07.
11. Mikolás Janota, Goetz Botterweck, Radu Grigore, and João P. Marques Silva. How to complete an interactive configuration process? In *Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 5901 of *LNCS*, pages 528–539. Springer, 2010.
12. Mikoláš Janota. Do SAT solvers make good configurators? In *First Workshop on Analyses of Software Product Lines (ASPL)*, September 2008.
13. Ulrich John and Ulrich Geske. Constraint-based configuration of large systems. In Oskar Bartenstein, Ulrich Geske, Markus Hannebauer, and Osama Yoshie, editors, *International Conference on Applications of Prolog (INAP 2001)*, volume 2543 of *LNCS*, pages 217–234. Springer, 2003.
14. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda). Feasibility study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, SW Engineering Institute, Carnegie Mellon University, 1990.

15. Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru. Mapping extended feature models to constraint logic programming over finite domains. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond (SPLC)*, volume 6287 of *LNCS*, pages 286–299. Springer, 2010.
16. R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.
17. R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. Springer, 2011.
18. Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2007.
19. Denny Schneeweiss. Grafische, interaktive Produktkonfiguration mit Finite-Domain-Constraints. Diplomarbeit, Brandenburgische Technische Universität Cottbus, 2011.
20. Sathiamoorthy Subbarayan. Integrating csp decomposition techniques and bdds for compiling configuration problems. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 3524 of *LNCS*, pages 351–365. Springer, 2005.
21. Sathiamoorthy Subbarayan, Rune M. Jensen, Tarik Hadzic, Henrik R. Andersen, Henrik Hulgaard, and Jesper Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Workshop on CSP Techniques with Immediate Application (CSPIA)*, pages 97–111, 2004.
22. Mitchell M. Tseng and Jianxin Jiao. Mass customization. In *Handbook of Industrial Engineering, Technology and Operation Management*. John Wiley & Sons, 3 edition, 2001.
23. Xtext. language development framework. http://www.eclipse.org/Xtext/. last visited 2011-07-07.
24. Hua Yan, Wei Zhang, Haiyan Zhao, and Hong Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In Stephen H. Edwards and Gregory Kulczycki, editors, *Conference on Software Reuse (ICSR)*, volume 5791 of *LNCS*, pages 65–75. Springer, 2009.

Each Normal Logic Program has a 2-valued Minimal Hypotheses Semantics

Alexandre Miguel Pinto and Luís Moniz Pereira
{amp|lmp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Abstract. In this paper we explore a unifying approach — that of hypotheses assumption — as a means to provide a semantics for all Normal Logic Programs (NLPs), the Minimal Hypotheses (MH) semantics [1]. This semantics takes a positive hypotheses assumption approach as a means to guarantee the desirable properties of model existence, relevance and cumulativity, and of generalizing the Stable Models semantics in the process. To do so we first introduce the fundamental semantic concept of minimality of assumed positive hypotheses, define the MH semantics, and analyze the semantics' properties and applicability. Indeed, abductive Logic Programming can be conceptually captured by a strategy centered on the assumption of abducibles (or hypotheses). Likewise, the Argumentation perspective of Logic Programs (e.g. [7]) also lends itself to an arguments (or hypotheses) assumption approach. Previous works on Abduction (e.g. [12]) have depicted the atoms of default negated literals in NLPs as abducibles, i.e., assumable hypotheses. We take a complementary and more general view than these works to NLP semantics by employing positive hypotheses instead.
Keywords: Hypotheses, Semantics, NLPs, Abduction, Argumentation.

1 Background

Logic Programs have long been used in Knowledge Representation and Reasoning.

Definition 1. ***Normal Logic Program.*** *By an alphabet $\mathcal{A}$ of a language $\mathcal{L}$ we mean (finite or countably infinite) disjoint sets of constants, predicate symbols, and function symbols, with at least one constant. In addition, any alphabet is assumed to contain a countably infinite set of distinguished variable symbols. A term over $\mathcal{A}$ is defined recursively as either a variable, a constant or an expression of the form $f(t_1, \ldots, t_n)$ where f is a function symbol of $\mathcal{A}$, n its arity, and the t_i are terms. An atom over $\mathcal{A}$ is an expression of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol of $\mathcal{A}$, and the t_i are terms. A literal is either an atom A or its default negation $not\ A$. We dub default literals (or default negated literals — DNLs, for short) those of the form $not\ A$. A term (resp. atom, literal) is said ground if it does not contain variables. The set of all ground terms (resp. atoms) of $\mathcal{A}$ is called the Herbrand universe (resp. base) of $\mathcal{A}$. For short we use $\mathcal{H}$ to denote the Herbrand base of $\mathcal{A}$. A Normal Logic Program (NLP) is a possibly infinite set of rules (with no infinite descending chains of syntactical dependency) of the form*

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m, \text{ (with } m, n \geq 0 \text{ and finite)}$$

where H, the B_i and the C_j are atoms, and each rule stands for all its ground instances. In conformity with the standard convention, we write rules of the form $H \leftarrow$ also simply as H (known as "facts"). An NLP P is called definite if none of its rules contain default literals. H is the head of the rule r, denoted by $head(r)$, and $body(r)$ denotes the set $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ of all the literals in the body of r.

When doing problem modelling with logic programs, rules of the form

$$\bot \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m, \text{ (with } m, n \geq 0 \text{ and finite)}$$

[1] This paper is a very condensed summary of some of the main contributions of the PhD Thesis [19] of the first author, supported by FCT-MCTES grant SFRH / BD / 28761 / 2006, and supervised by the second author.

with a non-empty body are known as a type of Integrity Constraints (ICs), specifically *denials*, and they are normally used to prune out unwanted candidate solutions. We abuse the 'not ' default negation notation applying it to non-empty sets of literals too: we write $not\ S$ to denote $\{not\ s : s \in S\}$, and confound $not\ not\ a \equiv a$. When S is an arbitrary, non-empty set of literals $S = \{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ we use

- S^+ denotes the set $\{B_1, \ldots, B_n\}$ of positive literals in S
- S^- denotes the set $\{not\ C_1, \ldots, not\ C_m\}$ of negative literals in S
- $|S| = S^+ \cup (not\ S^-)$ denotes the set $\{B_1, \ldots, B_n, C_1, \ldots, C_m\}$ of atoms of S

As expected, we say a set of literals S is consistent iff $S^+ \cap |S^-| = \emptyset$. We also write $heads(P)$ to denote the set of heads of non-IC rules of a (possibly constrained) program P, i.e., $heads(P) = \{head(r) : r \in P\} \setminus \{\bot\}$, and $facts(P)$ to denote the set of facts of P — $facts(P) = \{head(r) : r \in P \wedge body(r) = \emptyset\}$.

Definition 2. ***Part of body of a rule not in loop.*** *Let P be an NLP and r a rule of P. We write $\overline{body(r)}$ to denote the subset of $body(r)$ whose atoms do not depend on r. Formally, $\overline{body(r)}$ is the largest set of literals such that*

$$\overline{body(r)} \subseteq body(r) \wedge \forall_{a \in |\overline{body(r)}|} \nexists_{r_a \in P} (head(r_a) = a \wedge r_a \twoheadleftarrow r)$$

where $r_a \twoheadleftarrow r$ means rule r_a depends on rule r, i.e., either $head(r) \in |body(r_a)|$ or there is some other rule $r' \in P$ such that $r_a \twoheadleftarrow r'$ and $head(r) \in |body(r')|$.

Definition 3. ***Layer Supported and Classically supported interpretations.*** *We say an interpretation I of an NLP P is layer (classically) supported iff every atom a of I is layer (classically) supported in I. a is layer (classically) supported in I iff there is some rule r in P with $head(r) = a$ such that $I \models \overline{body(r)}$ ($I \models body(r)$). Likewise, we say the rule r is layer (classically) supported in I iff $I \models \overline{body(r)}$ ($I \models body(r)$).*

Literals in $\overline{body(r)}$ are, by definition, not in loop with r. The notion of layered support requires that all such literals be *true* under I in order for $head(r)$ to be layer supported in I. Hence, if $\overline{body(r)}$ is empty, $head(r)$ is *ipso facto* layer supported.

Proposition 1. ***Classical Support implies Layered Support.*** *Given a NLP P, an interpretation I, and an atom a such that $a \in I$, if a is classically supported in I then a is also layer supported in I.*

Proof. Knowing that, by definition, $\overline{body(r)} \subseteq body(r)$ for every rule r, it follows trivially that a is layer supported in I if a is classically supported in I. □

2 Motivation

"Why the need for another 2-valued semantics for NLPs since we already have the Stable Models one?" The question has its merit since the Stable Models (SMs) semantics [9] is exactly what is necessary for so many problem solving issues, but the answer to it is best understood when we ask it the other way around: "Is there any situation where the SMs semantics does not provide all the intended models?" and "Is there any 2-valued generalization of SMs that keeps the intended models it does provide, adds the missing intended ones, and also enjoys the useful properties of guarantee of model existence, relevance, and cumulativity?"

Example 1. **A Joint Vacation Problem — Merging Logic Programs.** Three friends are planning a joint vacation. First friend says "If we don't go to the mountains, then we should go to the beach". The second friend says "If we don't go to travelling, then we should go to the mountains". The third friend says "If we don't go to the beach, then we should go travelling". We code this information as the following NLP:

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach \end{aligned}$$

Each of these individual consistent rules come from a different friend. According to the SMs, each friend had a "solution" (a SM) for his own rule, but when we put the three rules together, because they form an odd loop over negation (OLON), the resulting merged logic program has no SM. If we assume $beach$ is *true* then we cannot conclude $travel$ and therefore we conclude $mountain$ is also *true* — this gives rise to the $\{beach, mountain, not\ travel\}$ joint and multi-place vacation solution. The other (symmetric) two are $\{mountain, not\ beach, travel\}$ and $\{travel, not\ mountain,$
$beach\}$. This example too shows the importance of having a 2-valued semantics guaranteeing model existence, in this case for the sake of arbitrary merging of logic programs (and for the sake of existence of a joint vacation for these three friends).

Increased Declarativity. An IC is a rule whose head is $\bot$, and although such syntactical definition of IC is generally accepted as standard, the SM semantics can employ odd loops over negation, such as the $a \leftarrow not\ a, X$ to act as ICs, thereby mixing and unnecessarily confounding two distinct Knowledge Representation issues: the one of IC use, and the one of assigning semantics to loops. For the sake of declarativity, rules with $\bot$ head should be the only way to write ICs in a LP: no rule, or combination of rules, with head different from $\bot$ should possibly act as IC(s) under any given semantics. It is commonly argued that answer sets (or stable models) of a program correspond to the solutions of the corresponding problem, so no answer set means no solution. We argue against this position: "normal" logic rules (i.e., non-ICs) should be used to shape the candidate-solution space, whereas ICs, and ICs alone, should be allowed to play the role of cutting down the undesired candidate-solutions. In this regard, an IC-free NLP should always have a model; if some problem modelled as an NLP with ICs has no solution (i.e., no model) that should be due only to the ICs, not to the "normal" rules.

Argumentation From an argumentation perspective, the author of [7], states:

> *"Stable extensions do not capture the intuitive semantics of every meaningful argumentation system."*

where the "stable extensions" have a 1-to-1 correspondence to the SMs ([7]), and also

> *"Let P be a knowledge base represented either as a logic program, or as a nonmonotonic theory or as an argumentation framework. Then there is not necessarily a "bug" in P if P has no stable semantics.*
>
> *This theorem defeats an often held opinion in the logic programming and nonmonotonic reasoning community that if a logic program or a nonmonotonic theory has no stable semantics then there is something "wrong" in it."*

Thus, a criterion different from the *stability* one must be used in order to effectively model every argumentation framework adequately.

Arbitrary Updates and/or Merges One of the main goals behind the conception of non-monotonic logics was the ability to deal with the changing, evolving, updating of knowledge. There are scenarios where it is possible and useful to combine several Knowledge Bases (possibly from different authors or sources) into a single one, and/or to update a given KB with new knowledge. Assuming the KBs are coded as IC-free NLPs, as well as the updates, the resulting KB is also an IC-free NLP. In such a case, the resulting (merged and/or updated) KB should always have a semantics. This should be true particularly in the case of NLPs where no negations are allowed in the heads of rules. In this case no contradictions can arise because there are no conflicting rule heads. The lack of such guarantee when the underlying semantics used is the Stable Models, for example, compromises the possibility of arbitrarily updating and/or merging KBs (coded as IC-free NLPs). In the case of self-updating programs, the desirable "liveness" property is put into question, even without outside intervention.

These motivational issues raise the questions "Which should be the 2-valued models of an NLP when it has no SMs?", "How do these relate to SMs?", "Is there a uniform approach to characterize both such models and the SMs?", and "Is there any 2-valued generalization of the SMs that encompasses the intuitive semantics of *every* logic program?". Answering such questions is a paramount motivation and thrust in this paper.

2.1 Intuitively Desired Semantics

It is commonly accepted that the non-stratification of the default not is the fundamental ingredient which allows for the possibility of existence of several models for a program. The non-stratified DNLs (i.e., in a loop) of a program can thus be seen as non-deterministically assumable choices. The rules in the program, as well as the particular semantics we wish to assign them, is what constrains which sets of those choices we take as acceptable. Programs with OLONs (ex. 1) are said to be "contradictory" by the SMs community because the latter takes a negative hypotheses assumption approach, consistently maximizing them, i.e., DNLs are seen as assumable/abducible hypotheses. In ex.1 though, assuming whichever maximal negative hypotheses leads to a positive contradictory conclusion via the rules. On the other hand, if we take a consistent minimal *positive* hypotheses assumption (where the assumed hypotheses are the *atoms* of the DNLs), then it is impossible to achieve a contradiction since no negative conclusions can be drawn from NLP rules. Minimizing positive assumptions implies the maximizing of negative ones but gaining an extra degree of freedom.

2.2 Desirable Formal Properties

Only ICs (rules with $\bot$ head) should "endanger" model existence in a logic program. Therefore, a semantics for NLPs with no ICs should guarantee model existence (which, e.g., does not occur with SMs). Relevance is also a useful property since it allows the development of top-down query-driven proof-procedures that allow for the sound and complete search for answers to a user's query. This is useful in the sense that in order to find an answer to a query only the relevant part of the program must be considered, whereas with a non-relevant semantics the whole program must be considered, with corresponding performance disadvantage compared to a relevant semantics.

Definition 4. ***Relevant part of*** P ***for atom*** a**.** *The relevant part of NLP* P *for atom* a *is* $Rel_P(a) = \{r_a \in P : head(r_a) = a\} \cup \{r \in P : \exists_{r_a \in P \wedge head(r_a)=a} r_a \twoheadleftarrow r\}$

Definition 5. ***Relevance (adapted from [5]).*** *A semantics* Sem *for logic programs is said Relevant iff for every program* P

$$\forall_{a \in \mathcal{H}_P}(\forall_{M \in Models_{Sem}(P)} a \in M) \Leftrightarrow (\forall_{M_a \in Models_{Sem}(Rel_P(a))} a \in M_a)$$

Moreover, cumulativity also plays a role in performance enhancement in the sense that only a semantics enjoying this property can take advantage of storing intermediate lemmas to speed up future computations.

Definition 6. ***Cumulativity (adapted from [4]).*** *Let* P *be an NLP, and* a, b *two atoms of* $\mathcal{H}_P$*. A semantics* Sem *is Cumulative iff the semantics of* P *remains unchanged when any atom* true *in the semantics is added to* P *as a fact:*

$$\forall_{a,b \in \mathcal{H}_P} \big((\forall_{M \in Models_{Sem}(P)} a \in M) \Rightarrow$$
$$(\forall_{M \in Models_{Sem}(P)} b \in M \Leftrightarrow \forall_{M_a \in Models_{Sem}(P \cup \{a\})} b \in M_a)\big)$$

Finally, each individual SM of a program, by being minimal and classically supported, should be accepted as a model according to every 2-valued semantics, and hence every 2-valued semantics should be a model conservative extension of Stable Models.

3 Syntactic Transformations

It is commonly accepted that definite LPs (i.e., without default negation) have only one 2-valued model — its *least model* which coincides with the Well-Founded Model (WFM [8]). This is also the case for locally stratified LPs. In such cases we can use a syntactic transformation on a program to obtain that model. In [2] the author defined the program Remainder (denoted by $\widehat{P}$) for calculating the WFM, which coincides with the unique perfect model for locally stratified LPs. The Remainder can thus be seen as a generalization for NLPs of the $lfp(T)$, the latter obtainable only from the subclass of definite LPs. We recap here the definitions necessary for the Remainder because we will use it in the definition of our Minimal Hypotheses

semantics. The intuitive gist of MH semantics (formally defined in section 4) is as follows: an interpretation M_H is a MH model of program P iff there is some minimal set of hypotheses H such that the truth-values of all atoms of P become determined assuming the atoms in H as *true*. We resort to the program Remainder as a deterministic (and efficient, i.e., computable in polynomial time) means to find out if the truth-values of all literals became determined or not — we will see below how the Remainder can be used to find this out.

3.1 Program Remainder

For self-containment, we include here the definitions of [2] upon which the Remainder relies, and adapt them where convenient to better match the syntactic conventions used throughout this paper.

Definition 7. ***Program transformation (def. 4.2 of [2]).*** *A* program transformation *is a relation $\mapsto$ between ground logic programs. A semantics S allows a transformation $\mapsto$ iff $Models_S(P_1) = Models_S(P_2)$ for all P_1 and P_2 with $P_1 \mapsto P_2$. We write $\mapsto^*$ to denote the fixed point of the $\mapsto$ operation, i.e., $P \mapsto^* P'$ where $\nexists_{P'' \neq P'} P' \mapsto P''$. It follows that $P \mapsto^* P' \Rightarrow P' \mapsto P'$.*

Definition 8. ***Positive reduction (def. 4.6 of [2]).*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* positive reduction *$(P_1 \mapsto_P P_2)$ iff there is a rule $r \in P_1$ and a negative literal $not\ b \in body(r)$ such that $b \notin heads(P_1)$, i.e., there is no rule for b in P_1, and $P_2 = (P_1 \setminus \{r\}) \cup \{head(r) \leftarrow (body(r) \setminus \{not\ b\})\}$.*

Definition 9. ***Negative reduction (def. 4.7 of [2]).*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* negative reduction *$(P_1 \mapsto_N P_2)$ iff there is a rule $r \in P_1$ and a negative literal $not\ b \in body(r)$ such that $b \in facts(P_1)$, i.e., b appears as a fact in P_1, and $P_2 = P_1 \setminus \{r\}$.*

Negative reduction is consistent with classical support, but not with the layered one. Therefore, we introduce now a layered version of the negative reduction operation.

Definition 10. ***Layered negative reduction.*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* layered negative reduction *$(P_1 \mapsto_{LN} P_2)$ iff there is a rule $r \in P_1$ and a negative literal $not\ b \in \overline{body(r)}$ such that $b \in facts(P_1)$, i.e., b appears as a fact in P_1, and $P_2 = P_1 \setminus \{r\}$.*

The Strongly Connected Components (SCCs) of rules of a program can be calculated in polynomial time [20]. Once the SCCs of rules have been identified, the $\overline{body(r)}$ subset of $body(r)$, for each rule r, is identifiable in linear time — one needs to check just once for each literal in $body(r)$ if it is also in $\overline{body(r)}$. Therefore, these polynomial time complexity operations are all the added complexity Layered negative reduction adds over regular Negative reduction.

Definition 11. ***Success (def. 5.2 of [2]).*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* success *$(P_1 \mapsto_S P_2)$ iff there are a rule $r \in P_1$ and a fact $b \in facts(P_1)$ such that $b \in body(r)$, and $P_2 = (P_1 \setminus \{r\}) \cup \{head(r) \leftarrow (body(r) \setminus \{b\})\}$.*

Definition 12. ***Failure (def. 5.3 of [2]).*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* failure *$(P_1 \mapsto_F P_2)$ iff there are a rule $r \in P_1$ and a positive literal $b \in body(r)$ such that $b \notin heads(P_1)$, i.e., there are no rules for b in P_1, and $P_2 = P_1 \setminus \{r\}$.*

Definition 13. ***Loop detection (def. 5.10 of [2]).*** *Let P_1 and P_2 be ground programs. Program P_2 results from P_1 by* loop detection *$(P_1 \mapsto_L P_2)$ iff there is a set $\mathcal{A}$ of ground atoms such that*

1. *for each rule $r \in P_1$, if $head(r) \in \mathcal{A}$, then $body(r) \cap \mathcal{A} \neq \emptyset$,*
2. *$P_2 := \{r \in P_1 | body(r) \cap \mathcal{A} = \emptyset\}$,*
3. *$P_1 \neq P_2$.*

We are not entering here into the details of the *loop detection* step, but just taking note that 1) such a set $\mathcal{A}$ corresponds to an unfounded set (cf. [8]); 2) loop detection is computationally equivalent to finding the SCCs [20], and is known to be of polynomial time complexity; and 3) the atoms in the unfounded set $\mathcal{A}$ have all their corresponding rules involved in SCCs where all heads of rules in loop appear positive in the bodies of the rules in loop.

Definition 14. ***Reduction (def. 5.15 of [2]).***
Let $\mapsto_X$ *denote the rewriting system:* $\mapsto_X := \mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

Definition 15. ***Layered reduction.***
Let $\mapsto_{LX}$ *denote the rewriting system:* $\mapsto_{LX} := \mapsto_P \cup \mapsto_{LN} \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

Definition 16. ***Remainder (def. 5.17 of [2]).*** *Let* P *be a program. Let* $\widehat{P}$ *satisfy* $ground(P) \mapsto_X^* \widehat{P}$. *Then* $\widehat{P}$ *is called the* remainder *of* P, *and is guaranteed to exist and to be unique to* P. *Moreover, the calculus of* $\mapsto_X^*$ *is known to be of polynomial time complexity [2]. When convenient, we write* $Rem(P)$ *instead of* $\widehat{P}$.

An important result from [2] is that the WFM of P is such that $WFM^+(P) = facts(\widehat{P})$, $WFM^{+u} = heads(\widehat{P})$, and $WFM^-(P) = \mathcal{H}_P \setminus WFM^{+u}(P)$, where $WFM^+(P)$ denotes the set of atoms of P *true* in the WFM, $WFM^{+u}(P)$ denotes the set of atoms of P *true* or *undefined* in the WFM, and $WFM^-(P)$ denotes the set of atoms of P *false* in the WFM.

Definition 17. ***Layered Remainder.*** *Let* P *be a program. Let the program* $\mathring{P}$ *satisfy* $ground(P) \mapsto_{LX}^* \mathring{P}$. *Then* $\mathring{P}$ *is called a* layered remainder *of* P. *Since* $\mathring{P}$ *is equivalent to* $\widehat{P}$, *apart from the difference between* $\mapsto_{LN}$ *and* $\mapsto_N$, *it is trivial that* $\mathring{P}$ *is also guaranteed to exist and to be unique for* P. *Moreover, the calculus of* $\mapsto_{LX}^*$ *is likewise of polynomial time complexity because* $\mapsto_{LN}$ *is also of polynomial time complexity.*

The remainder's rewrite rules are provably confluent, ie. independent of application order. The layered remainder's rules differ only in the negative reduction rule and the confluence proof of the former is readily adapted to the latter.

Example 2. $\mathring{P}$ **versus** $\widehat{P}$**.** Recall the program from example 1 but now with an additional fourth stubborn friend who insists on going to the beach no matter what. $P =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach \\ beach & \end{aligned}$$

We can clearly see that the single fact rule does not depend on any other, and that the remaining three rules forming the loop all depend on each other and on the fact rule $beach$. $\widehat{P}$ is the fixed point of $\mapsto_X$, i.e., the fixed point of $\mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$. Since $beach$ is a fact, the $\mapsto_N$ transformation deletes the $travel \leftarrow not\ beach$ rule; i.e., $P \mapsto_N P'$ is such that

$P' = \{beach \leftarrow not\ mountain \quad mountain \leftarrow not\ travel \quad beach \leftarrow\}$

Now in P' there are no rules for $travel$ and hence we can apply the $\mapsto_P$ transformation which deletes the $not\ travel$ from the body of $mountain$'s rule; i.e, $P' \mapsto_P P''$ where $P'' = \{beach \leftarrow not\ mountain$ $mountain \leftarrow \quad beach \leftarrow\}$

Finally, in P'' $mountain$ is a fact and hence we can again apply the $\mapsto_N$ obtaining $P'' \mapsto_P P'''$ where $P''' = \{mountain \leftarrow \quad beach \leftarrow\}$ upon which no more transformations can be applied, so $\widehat{P} = P'''$. Instead, $\mathring{P} = P$ is the fixed point of $\mapsto_{LX}$, i.e., the fixed point of $\mapsto_P \cup \mapsto_{LN} \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

4 Minimal Hypotheses Semantics

4.1 Choosing Hypotheses

The abductive perspective of [12] depicts the atoms of DNLs as abducibles, i.e., assumable hypotheses. Atoms of DNLs can be considered as abducibles, i.e., assumable hypotheses, but not all of them. When we have a locally stratified program we cannot really say there is any degree of freedom in assuming truth values for the atoms of the program's DNLs. So, we realize that only the atoms of DNLs involved in SCCs[2] are eligible to be considered further assumable hypotheses.

[2] Strongly Connected Components, as in Examples 1 and 2

Both the SMs and the approach of [12], when taking the abductive perspective, adopt negative hypotheses only. This approach works fine for some instances of non-well-founded negation such as loops (in particular, for even loops over negation like this one), but not for odd loops over negation like, e.g. $a \leftarrow not\ a$: assuming $not\ a$ would lead to the conclusion that a is *true* which contradicts the initial assumption. To overcome this problem, we generalized the hypotheses assumption perspective to allow the adoption, not only of negative hypotheses, but also of positive ones. Having taken this generalization step we realized that positive hypotheses assumption alone is sufficient to address all situations, i.e., there is no need for both positive and negative hypotheses assumption. Indeed, because we minimize the positive hypotheses we are with one stroke maximizing the negative ones, which has been the traditional way of dealing with the CWA, and also with stable models because the latter's requirement of classical support minimizes models.

In example 1 we saw three solutions, each assuming as *true* one of the DNLs in the loop. Adding a fourth stubborn friend insisting on going to the beach, as in example 2, should still permit the two solutions $\{beach, mountain, not\ travel\}$ and $\{travel, not\ mountain, beach\}$. The only way to permit both these solutions is by resorting to the Layered Remainder, and not to the Remainder, as a means to identify the set of assumable hypotheses.

Thus, all the literals of P that are not determined *false* in $\mathring{P}$ are candidates for the role of hypotheses we may consider to assume as *true*. Merging this perspective with the abductive perspective of [12] (where the DNLs are the abducibles) we come to the following definition of the Hypotheses set of a program.

Definition 18. ***Hypotheses set of a program.*** *Let P be an NLP. We write $Hyps(P)$ to denote the set of assumable hypotheses of P: the atoms that appear as DNLs in the bodies of rules of $\mathring{P}$. Formally, $Hyps(P) = \{a : \exists_{r \in \mathring{P}} not\ a \in body(r)\}$.*

One can define a classical support compatible version of the Hypotheses set of a program, only using to that effect the Remainder instead of the Layered Remainder. I.e.,

Definition 19. ***Classical Hypotheses set of a program.*** *Let P be an NLP. We write $CHyps(P)$ to denote the set of assumable hypotheses of P consistent with the classical notion of support: the atoms that appear as DNLs in the bodies of rules of $\widehat{P}$. Formally, $CHyps(P) = \{a : \exists_{r \in \widehat{P}} not\ a \in body(r)\}$.*

Here we take the layered support compatible approach and, therefore, we will use the Hypotheses set as in definition 18. Since $CHyps(P) \subseteq Hyps(P)$ for every NLP P, there is no generality loss in using $Hyps(P)$ instead of $CHyps(P)$, while using $Hyps(P)$ allows for some useful semantics properties examined in the sequel.

4.2 Definition

Intuitively, a Minimal Hypotheses model of a program is obtained from a minimal set of hypotheses which is sufficiently large to determine the truth-value of all literals via Remainder.

Definition 20. ***Minimal Hypotheses model.*** *Let P be an NLP. Let $Hyps(P)$ be the set of assumable hypotheses of P (cf. definition 18), and H some subset of $Hyps(P)$.*

A 2-valued model M of P is a Minimal Hypotheses model of P iff

$$M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$$

where $H = \emptyset$ or H is non-empty set-inclusion minimal (the set-inclusion minimality is considered only for non-empty Hs). I.e., the hypotheses set H is minimal but sufficient to determine (via Remainder) the truth-value of all literals in the program.

We already know that $WFM^+(P) = facts(\widehat{P})$ and that $WFM^{+u}(P) = heads(\widehat{P})$. Thus, whenever $facts(\widehat{P}) = heads(\widehat{P})$ we have $WFM^+(P) = WFM^{+u}(P)$ which means $WFM^u(P) = \emptyset$. Moreover, whenever $WFM^u(P) = \emptyset$ we know, by Corollary 5.6 of [8], that the 2-valued model M such that $M^+ = facts(\widehat{P})$ is the unique stable model of P. Thus, we conclude that, as an alternative equivalent definition,

M is a Minimal Hypotheses model of P iff M is a stable model of $P \cup H$ where H is empty or a non-empty set-inclusion minimal subset of $Hyps(P)$. Moreover, it follows immediately that every SM of P is a Minimal Hypotheses model of P.

In example 2 we can thus see that we have the two models $\{beach, mountain,$
$not\ travel\}$ and $\{travel, beach, not\ mountain\}$. This is the case because the addition of the fourth stubborn friend does not change the set of $Hyps(P)$ which is based upon the Layered Remainder, and not on the Remainder.

Example 3. **Minimal Hypotheses models for the vacation with passport variation.** Consider again the vacation problem from example 1 with a variation including the need for valid passports for travelling $P =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

We have $P = \mathring{P} = \widehat{P}$ and thus $Hyps(P) = \{beach, mountain, travel, passport\_ok, expired\_passport\}$.
Let us see which are the MH models for this program.
$H = \emptyset$ does not yield a MH model.
Assuming $H = \{beach\}$ we have $P \cup H = P \cup \{beach\} =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ beach & \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

and $\widehat{P \cup H} =$

$$\begin{aligned} mountain & \\ beach & \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \end{aligned}$$

which means $H = \{beach\}$ is not sufficient to determine the truth values of all literals of P. One can easily see that the same happens for $H = \{mountain\}$ and for $H = \{travel\}$: in either case the literals $passport\_ok$ and $expired\_passport$ remain non-determined.
If we assume $H = \{expired\_passport\}$ then $P \cup H$ is

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \\ expired\_passport & \end{aligned}$$

and $\widehat{P \cup H} =$

$$\begin{aligned} mountain & \\ expired\_passport & \end{aligned}$$

which means $M^+_{expired\_passport} = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H}) = \{mountain, expired\_passport\}$, i.e.,
$M_{expired\_passport} = \{not\ beach, mountain, not\ travel, not\ passport\_ok, expired\_passport\}$, is a MH

model of P. Since assuming $H = \{expired\_passport\}$ alone is sufficient to determine all literals, there is no other set of hypotheses H' of P such that $H' \supset \{expired\_passport\}$ (notice the *strict* $\supset$, not $\supseteq$), yielding a MH model of P. E.g., $H' = \{travel, expired\_passport\}$ does not lead to a MH model of P simply because H' is not minimal w.r.t. $H = \{expired\_passport\}$.
If we assume $H = \{passport\_ok\}$ then $P \cup H$ is

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach, not\ expired\_passport \\ \\ passport\_ok &\leftarrow not\ expired\_passport \\ expired\_passport &\leftarrow not\ passport\_ok \\ passport\_ok & \end{aligned}$$

and $\widehat{P \cup H} =$

$$\begin{aligned} beach &\leftarrow not\ mountain \\ mountain &\leftarrow not\ travel \\ travel &\leftarrow not\ beach \\ passport\_ok & \end{aligned}$$

which, apart from the fact $passport\_ok$, corresponds to the original version of this example and still leaves literals with non-determined truth-values. I.e., assuming the passports are OK allows for the three possibilities of example 1 but it is not enough to entirely "solve" the vacation problem: we need some hypotheses set containing one of $beach$, $mountain$, or $travel$ if (in this case, *and only if*) it also contains $passport\_ok$.

Example 4. **Minimality of Hypotheses does not guarantee minimality of model.** Let P, with no SMs, be

$$\begin{aligned} a &\leftarrow not\ b, c \\ b &\leftarrow not\ c, not\ a \\ c &\leftarrow not\ a, b \end{aligned}$$

In this case $P = \widehat{P} = \mathring{P}$, which makes $Hyps(P) = \{a, b, c\}$.
$H = \emptyset$ does not determine all literals of P because $facts(\widehat{P \cup \emptyset}) = facts(\widehat{P}) = \emptyset$ and $heads(\widehat{P \cup \emptyset}) = heads(\widehat{P}) = \{a, b, c\}$.
$H = \{a\}$ does determine all literals of P because $facts(\widehat{P \cup \{a\}}) = \{a\}$ and $heads(\widehat{P \cup \{a\}}) = \{a\}$, thus yielding the MH model M_a such that $M_a^+ = facts(\widehat{P \cup \{a\}}) = \{a\}$, i.e., $M_a = \{a, not\ b, not\ c\}$.
$H = \{c\}$ is also a minimal set of hypotheses determining all literals because $facts(\widehat{P \cup \{c\}}) = \{a, c\}$ and $heads(\widehat{P \cup \{c\}}) = \{a, c\}$, thus yielding the MH model M_c of P such that $M_c^+ = facts(\widehat{P \cup \{c\}}) = \{a, c\}$, i.e., $M_c = \{a, not\ b, c\}$. However, M_c is not a minimal model of P because $M_c^+ = \{a, c\}$ is a strict superset of $M_a^+ = \{a\}$. M_c is indeed an MH model of P, but just not a minimal model thereby being a clear example of how minimality of hypotheses does not entail minimality of consequences. Just to make this example complete, we show that $H = \{b\}$ also determines all literals of P because $facts(\widehat{P \cup \{b\}}) = \{b, c\}$ and $heads(\widehat{P \cup \{b\}}) = \{b, c\}$, thus yielding the MH model M_b such that $M_b^+ = facts(\widehat{P \cup \{b\}}) = \{b, c\}$, i.e., $M_b = \{not\ a, b, c\}$. Any other hypotheses set is necessarily a strict superset of either $H = \{a\}$, $H = \{b\}$, or $H = \{c\}$ and, therefore, not set-inclusion minimal; i.e., there are no more MH models of P.

Also, not all minimal models of a program are MH models, as the following example shows.

Example 5. **Some minimal models are not Minimal Hypotheses models.** Let P (with no SMs) be

$$\begin{aligned} a &\leftarrow k \\ k &\leftarrow not\ t \\ t &\leftarrow a, b \\ a &\leftarrow not\ b \\ b &\leftarrow not\ a \end{aligned}$$

In this case $P = \widehat{P} = \mathring{P}$ and therefore $Hyps(P) = \{a, b, t\}$. Since $facts(\widehat{P}) \neq heads(\widehat{P})$, the hypotheses set $H = \emptyset$ does not yield a MH model. Assuming $H = \{a\}$ we have $\widehat{P \cup H} = \widehat{P \cup \{a\}} = \{a \leftarrow \quad, \quad k \leftarrow\}$ so, $\widehat{P \cup H}$ is the set of facts $\{a, k\}$ and, therefore, M_a such that $M_a^+ = facts(\widehat{P \cup H}) = facts(\widehat{P \cup \{a\}}) = \{a, k\}$, is a MH model of P. Assuming $H = \{b\}$ we have $\widehat{P \cup \{b\}} =$

$$\begin{array}{l} a \leftarrow k \\ k \leftarrow not\ t \\ t \leftarrow a \\ b \leftarrow not\ a \\ b \end{array}$$

thus $facts(\widehat{P \cup \{b\}}) = \{b\} \neq heads(\widehat{P \cup \{b\}}) = \{a, b, t, k\}$, which means the set of hypotheses $H = \{b\}$ does not yield a MH model of P. Assuming $H = \{t\}$ we have $\widehat{P \cup \{t\}} =$

$$\begin{array}{l} t \leftarrow a, b \\ b \leftarrow not\ a \\ a \leftarrow not\ b \\ t \end{array}$$

thus $facts(\widehat{P \cup \{t\}}) = \{t\} \neq heads(\widehat{P \cup \{t\}}) = \{a, b, t\}$, which means the set of hypotheses $H = \{t\}$ does not yield a MH model of P.
Since we already know that $H = \{a\}$ yields an MH model M_a with $M_a^+ = \{a, k\}$, there is no point in trying out any subset H' of $Hyps(P) = \{a, b, t\}$ such that $a \in H'$ because any such subset would not be minimal w.r.t. $H = \{a\}$. Let us, therefore, move on to the unique subset left: $H = \{b, t\}$. Assuming $H = \{b, t\}$ we have $\widehat{P \cup \{b, t\}} = \{t \leftarrow \quad, \quad b \leftarrow\}$ thus $facts(\widehat{P \cup \{b, t\}}) = \{b, t\} = heads(\widehat{P \cup \{b, t\}})$, which means $M_{b,t}$ such that $M_{b,t}^+ = facts(\widehat{P \cup H}) = facts(\widehat{P \cup \{b, t\}}) = \{b, t\}$, is a MH model of P.
It is important to remark that this program has other classical models, e.g, $\{a, k\}, \{b, t\}$, and $\{a, t\}$, but only the first two are Minimal Hypotheses models — $\{a, t\}$ is obtainable only via the set of hypotheses $\{a, t\}$ which is non-minimal w.r.t. $H = \{a\}$ that yields the MH model $\{a, k\}$.

4.3 Properties

The minimality of H is not sufficient to ensure minimality of $M^+ = facts(\widehat{P \cup H})$ making its checking explicitly necessary if that is so desired. Minimality of hypotheses is indeed the common practice is science, not the minimality of their inevitable consequences. To the contrary, the more of these the better because it signifies a greater predictive power.

In Logic Programming model minimality is a consequence of definitions: the T operator in definite programs is conducive to defining a least fixed point, a unique minimal model semantics; in SM, though there may be more than one model, minimality turns out to be a property because of the stability (and its attendant classical support) requirement; in the WFS, again the existence of a least fixed point operator affords a minimal (information) model. In abduction too, minimality of consequences is not a caveat, but rather minimality of hypotheses is, if that even. Hence our approach to LP semantics via MHS is novel indeed, and insisting instead on positive hypotheses establishes an improved and more general link to abduction and argumentation [16, 17].

Theorem 1. ***At least one Minimal Hypotheses model of*** P ***complies with the Well-Founded Model.*** *Let P be an NLP. Then, there is at least one Minimal Hypotheses model M of P such that $M^+ \supseteq WFM^+(P)$ and $M^+ \subseteq WFM^{+u}(P)$.*

Proof. If $facts(\widehat{P}) = heads(\widehat{P})$ or equivalently, $WFM^u(P) = \emptyset$, then M_H is a MH model of P given that $H = \emptyset$ because $M_H^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H}) = facts(\widehat{P \cup \emptyset}) = heads(\widehat{P \cup \emptyset}) = facts(\widehat{P}) = heads(\widehat{P})$. On the other hand, if $facts(\widehat{P}) \neq heads(\widehat{P})$, then there is at least one non-empty set-inclusion minimal set of hypotheses $H \subseteq Hyps(P)$ such that $H \supseteq facts(P)$. The corresponding M_H is, by definition, a MH model of P which is guaranteed to comply with $M_H^+ \supseteq WFM^+(P) = facts(\widehat{P})$ and $M_H^- \supseteq not\ WFM^-(P) = not\ (\mathcal{H}_P \setminus M_H^+)$. □

Theorem 2. ***Minimal Hypotheses semantics guarantees model existence.*** *Let P be an NLP. There is always, at least, one Minimal Hypotheses model of P.*

Proof. It is trivial to see that one can always find a set $H \subseteq Hyps(P)$ such that $M^+_{H'} = facts(\widehat{P \cup H'}) = heads(\widehat{P \cup H'})$ — in the extreme case, $H' = Hyps(P)$. From such H' one can always select a minimal subset $H \subset H$ such that $M^+_H = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ still holds. □

4.4 Relevance

Theorem 3. ***Minimal Hypotheses semantics enjoys Relevance.*** *Let P be an NLP. Then, by definition 5, it holds that*

$$(\forall_{M \in Models_{MH}(P)} a \in M^+) \Leftrightarrow (\forall_{M_a \in Models_{MH}(Rel_P(a))} a \in M^+_a)$$

Proof. $\Rightarrow$: Assume $\forall_{M \in Models_{MH}(P)} a \in M^+$. Now we need to prove $\forall_{M_a \in Models_{MH}(Rel_P(a))} a \in M^+_a$. Assume some $M_a \in Models_{MH}(Rel_P(a))$; now we show that assuming $a \notin M^+_a$ leads to an absurdity. Since M_a is a 2-valued complete model of $Rel_P(a)$ we know that $|M_a| = \mathcal{H}_{Rel_P(a)}$ hence, if $a \notin M_a$, then necessarily $not\ a \in M^-_a$. Since $P \supseteq Rel_P(a)$, by theorem 2 we know that there is some model M' of P such that $M' \supseteq M_a$, and thus $not\ a \in M'^-$ which contradicts the initial assumption that $\forall_{M \in Models_{MH}(P)} a \in M^+$. We conclude $a \notin M_a$ cannot hold, i.e., $a \in M_a$ must hold. Since $a \in M^+$ hold for every model M of P, then $a \in M_a$ must hold for every model M_a of $Rel_P(a)$.

$\Leftarrow$: Assume $\forall_{M_a \in Models_{MH}(Rel_P(a))} a \in M^+_a$. Now we need to prove $\forall_{M \in Models_{MH}(P)} a \in M^+$. Let us write $P_{)a(}$ as an abbreviation of $P \setminus Rel_P(a)$. We have therefore $P = P_{)a(} \cup Rel_P(a)$. Let us now take $P_{)a(} \cup M_a$. We know that every NLP as an MH model, hence every MH model M of $P_{)a(} \cup M_a$ is such that $M \supseteq M_a$. Let H_{M_a} denote the Hypotheses set of M_a — i.e., $M^+_a = facts(\widehat{Rel_P(a) \cup H_{M_a}}) = heads(\widehat{Rel_P(a) \cup H_{M_a}})$, with $H_{M_a} = \emptyset$ or non-empty set-inclusion minimal, as per definition 20. If $facts(\widehat{P \cup H_{M_a}}) = heads(\widehat{P \cup H_{M_a}})$ then $M^+ = facts(\widehat{P \cup H_M}) = heads(\widehat{P \cup H_M})$ is an MH model of P with $H_M = H_{M_a}$ and, necessarily, $M \supseteq M_a$.

If $facts(\widehat{P \cup H_{M_a}}) \neq heads(\widehat{P \cup H_{M_a}})$ then, knowing that every program has a MH model, we can always find an MH model M of $P_{)a(} \cup M_a$, with $H' \subseteq Hyps(P_{)a(} \cup M_a)$, where $M^+ = facts(\widehat{P \cup H'}) = heads(\widehat{P \cup H'})$. Such M is thus $M^+ = facts(\widehat{P \cup H_M}) = heads(\widehat{P \cup H_M})$ where $H_M = H_{M_a} \cup H'$, which means M is a MH model of P with $M \supseteq M_a$. Since every model M_a of $Rel_P(a)$ is such that $a \in M^+_a$, then every model M of P must also be such that $a \in M$. □

4.5 Cumulativity

MH semantics enjoys Cumulativity thus allowing for lemma storing techniques to be used during computation of answers to queries.

Theorem 4. ***Minimal Hypotheses semantics enjoys Cumulativity.*** *Let P be an NLP. Then*

$$\forall_{a,b \in \mathcal{H}_\mathcal{P}} \big((\forall_{M \in Models_{MH}(\mathcal{P})} a \in M^+) \Rightarrow (\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+ \Leftrightarrow \forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M^+_a)\big)$$

Proof. Assume $\forall_{\substack{a \in \mathcal{H}_\mathcal{P} \\ M \in Models_{MH}(\mathcal{P})}} a \in M^+$.

$\Rightarrow$: Assume $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+$. Since every MH model M contains a it follows that all such M are also MH models of $P \cup \{a\}$. Since we assumed $b \in M$ as well, and we know that M is a MH model of $P \cup \{a\}$ we conclude b is also in those MH models M of $P \cup \{a\}$. By adding a as a fact we have necessarily $Hyps(P \cup \{a\}) \subseteq Hyps(P)$ which means that there cannot be more MH models for $P \cup \{a\}$ than for P. Since we already know that for every MH model M of P, M is also a MH model of $P \cup \{a\}$ we must conclude that $\forall_{M \in Models_{MH}(P)} \exists^1_{M' \in Models_{MH}(P \cup \{a\})}$ such that $M'^+ \supseteq M^+$. Since $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+$ we necessarily conclude $\forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M^+_a$.

$\Leftarrow$: Assume $\forall_{M_a \in Models_{MH}(\mathcal{P} \cup \{a\})} b \in M^+_a$. Since the MH semantics is relevant (theorem 3) if b does not depend on a then adding a as a fact to P or not has no impact on b's truth-value, and if $b \in M^+_a$ then

$b \in M^+$ as well. If b does depend on a, which is true in every MH model M of P, then either 1) b depends positively on a, and in this case since $a \in M$ then $b \in M$ as well; or 2) b depends negatively on a, and in this case the lack of a as a fact in P can only contribute, if at all, to make b true in M as well. Then we conclude $\forall_{M \in Models_{MH}(\mathcal{P})} b \in M^+$. □

4.6 Complexity

The complexity issues usually relate to a particular set of tasks, namely: 1) knowing if the program has a model; 2) if it has any model entailing some set of ground literals (a query); 3) if all models entail a set of literals. In the case of MH semantics, the answer to the first question is an immediate "yes" because MH semantics guarantees model existence for NLPs; the second and third questions correspond (respectively) to Brave and Cautious Reasoning, which we now analyse.

Brave Reasoning The complexity of the Brave Reasoning task with MH semantics, i.e., finding an MH model satisfying some particular set of literals is Σ_2^P-complete.

Theorem 5. ***Brave Reasoning with MH semantics is Σ_2^P-complete.*** *Let P be an NLP, and Q a set of literals, or* query. *Finding an MH model such that $M \supseteq Q$ is a Σ_2^P-complete task.*

Proof. To show that finding a MH model $M \supseteq Q$ is Σ_2^P-complete, note that a nondeterministic Turing machine with access to an NP-complete oracle can solve the problem as follows: nondeterministically guess a set H of hypotheses (i.e., a subset of $Hyps(P)$). It remains to check if H is empty or non-empty minimal such that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ and $M \supseteq Q$. Checking that $M^+ = facts(\widehat{P \cup H}) = heads(\widehat{P \cup H})$ can be done in polynomial time (because computing $\widehat{P \cup H}$ can be done in polynomial time [2] for whichever $P \cup H$), and checking H is empty or non-empty minimal requires a nondeterministic guess of a strict subset H' of H and then a polynomial check if $facts(\widehat{P \cup H'}) = heads(\widehat{P \cup H'})$. □

Cautious Reasoning Conversely, the Cautious Reasoning, i.e., guaranteeing that every MH model satisfies some particular set of literals, is Π_2^P-complete.

Theorem 6. ***Cautious Reasoning with MH semantics is Π_2^P-complete.*** *Let P be an NLP, and Q a set of literals, or* query. *Guaranteeing that all MH models are such that $M \supseteq Q$ is a Π_2^P-complete task.*

Proof. Cautious Reasoning is the complement of Brave Reasoning, and since the latter is Σ_2^P-complete (theorem 5), the former must necessarily be Π_2^P-complete. □

The set of hypotheses $Hyps(P)$ is obtained from $\mathring{P}$ which identifies rules that depend on themselves. The hypotheses are the atoms of DNLs of $\mathring{P}$, i.e., the "atoms of *not*s in loop". A Minimal Hypotheses model is then obtained from a minimal set of these hypotheses sufficient to determine the 2-valued truth-value of every literal in the program. The MH semantics imposes no ordering or preference between hypotheses — only their set-inclusion minimality. For this reason, we can think of the choosing of a set of hypotheses yielding a MH model as finding a minimal solution to a disjunction problem, where the disjuncts are the hypotheses. In this sense, it is therefore understandable that the complexity of the reasoning tasks with MH semantics is in line with that of, e.g., reasoning tasks with SM semantics with Disjunctive Logic Programs, i.e, Σ_2^P-complete and Π_2^P-complete.

In abductive reasoning (as well as in Belief Revision) one does not always require minimal solutions. Likewise, taking a hypotheses assumption based semantic approach, like the one of MH, one may not require minimality of assumed hypotheses. In such case, we would be under a non-Minimal Hypotheses semantics, and the complexity classes of the corresponding reasoning task would be one level down in the Polynomial Hierarchy relatively to the MH semantics, i.e., Brave Reasoning with a non-Minimal Hypotheses semantics would be NP-complete, and Cautious Reasoning would be coNP-complete. We leave the exploration of such possibilities for future work.

4.7 Comparisons

As we have seen all stable models are MH models. Since MH models are always guaranteed to exist for every NLP (cf. theorem 2) and SMs are not, it follows immediately that the Minimal Hypotheses semantics is a strict model conservative generalization of the Stable Models semantics. The MH models that are stable models are exactly those in which all rules are classically supported. With this criterion one can conclude whether some program does not have any stable models. For Normal Logic Programs, the Stable Models semantics coincides with the Answer-Set semantics (which is a generalization of SMs to Extended Logic Programs), where the latter is known (cf. [10]) to correspond to Reiter's default logic. Hence, all Reiter's default extensions have a corresponding Minimal Hypotheses model. Also, since Moore's expansions of an autoepistemic theory [13] are known to have a one-to-one correspondence with the stable models of the NLP version of the theory, we conclude that for every such expansion there is a matching Minimal Hypotheses model for the same NLP.

Disjunctive Logic Programs (DisjLPs — allowing for disjunctions in the heads of rules) can be syntactically transformed into NLPs by applying the Shifting Rule presented in [6] in all possible ways. By non-deterministically applying such transformation in all possible ways, several SCCs of rules may appear in the resulting NLP that were not present in the original DisjLP — assigning a meaning to every such SCC is a distinctive feature of MH semantics, unlike other semantics such as the SMs. This way, the MH semantics can be defined for DisjLPs as well: the MH models of a DisjLP are the MH models of the NLP resulting from the transformation via Shifting Rule.

There are other kinds of disjunction, like the one in logic programs with ordered disjunction (LPOD) [3]. These employ "*a new connective called ordered disjunction. The new connective allows to represent alternative, ranked options for problem solutions in the heads of rules*". As the author of [3] says "the semantics of logic programs with ordered disjunction is based on a preference relation on answer sets." This is different from the semantics assigned by MH since the latter includes no ordering, nor preferences, in the assumed minimal sets of hypotheses. E.g., in example 1 there is no notion of preference or ordering amongst candidate models — LPODs would not be the appropriate formalism for such cases. We leave for future work a thorough comparison of these approaches, namely comparing the semantics of LPODs against the MH models of LPODs transformed into NLPs (via the Shifting Rule).

The motivation for [21] is similar to our own — to assign a semantics to *every* NLP — however their approach is different from ours in the sense that the methods in [21] resort to contrapositive rules allowing any positive literal in the head to be shifted (by negating it) to the body or any negative literal in the body to be shifted to the head (by making it positive). This approach considers each rule as a disjunction making no distinction between such literals occurring in the rule, whether or not they are in loop with the head of the rule. This permits the shifting operations in [21] to create support for atoms that have no rules in the original program. E.g.

Example 6. **Nearly-Stable Models vs MH models.** Take the program $P =$

$$\begin{aligned} a &\leftarrow not\ b \\ b &\leftarrow not\ c \\ c &\leftarrow not\ a, not\ x \end{aligned}$$

According to the shifting operations in [21] this program could be transformed into $P' =$

$$\begin{aligned} b &\leftarrow not\ a \\ b &\leftarrow not\ c \\ x &\leftarrow not\ a, not\ c \end{aligned}$$

by shifting a and $not\ b$ in the first rule, and shifting the $not\ x$ to the head (becoming positive x) and c to the body (becoming negative $not\ c$) of the third rule thus allowing for $\{b, x\}$ (which is a stable model of P') to be a *nearly stable model* of P. In this sense the approach of [21] allows for the violation of the Closed-World Assumption. This does not happen with our approach: $\{b, x\}$ is not a Minimal Hypotheses model simply because since x has no rules in P it cannot be true in any MH model — $not\ x$ is not a member of $Hyps(P)$ (cf. def. 18).

As shown in theorem 1, at least one MH model of a program complies with its well-founded model, although not necessarily all MH models do. E.g., the program in Ex. 2 has the two MH models $\{beach, mountain, not\, travel\}$ and $\{beach, not\, mountain, travel\}$, whereas the $WFM(P)$imposes $WFM^{+}(P) = \{beach, mountain\}$, $WFM^{u}(P) = \emptyset$, and $WFM^{-}(P) = \{travel\}$. This is due to the set of Hypotheses $Hyps(P)$ of P being taken from $\mathring{P}$ (based on the layered support notion) instead of being taken from $\widehat{P}$ (based on the classical notion of support).

Not all Minimal Hypotheses models are Minimal Models of a program. The rationale behind MH semantics is minimality of hypotheses, but not necessarily minimality of consequences, the latter being enforceable, if so desired, as an additional requirement, although at the expense of increased complexity.

The relation between logic programs and argumentation systems has been considered for a long time now ([7] amongst many others) and we have also taken steps to understand and further that relationship [16–18]. Dung's Preferred Extensions [7] are maximal sets of negative hypotheses yielding consistent models. Preferred Extensions, however, these are not guaranteed to always yield 2-valued complete models. Our previous approaches [16, 17] to argumentation have already addressed the issue of 2-valued model existence guarantee, and the MH semantics also solves that problem by virtue of positive, instead of negative, hypotheses assumption.

5 Conclusions and Future Work

Taking a positive hypotheses assumption approach we defined the 2-valued Minimal Hypotheses semantics for NLPs that guarantees model existence, enjoys relevance and cumulativity, and is also a model conservative generalization of the SM semantics. Also, by adopting positive hypotheses, we not only generalized the argumentation based approach of [7], but the resulting MH semantics lends itself naturally to abductive reasoning, it being understood as hypothesizing plausible reasons sufficient for justifying given observations or supporting desired goals. We also defined the layered support notion which generalizes the classical one by recognizing the special role of loops.

For query answering, the MH semantics provides mainly three advantages over the SMs: 1) by enjoying Relevance top-down query-solving is possible, thereby circumventing whole model computation (and grounding) which is unavoidable with SMs; 2) by considering only the relevant sub-part of the program when answering a query it is possible to enact grounding of only those rules, if grounding is really desired, whereas with SM semantics whole program grounding is, once again, inevitable — grounding is known to be a major source of computational time consumption; MH semantics, by enjoying Relevance, permits curbing this task to the minimum sufficient to answer a query; 3) by enjoying Cumulativity, as soon as the truth-value of a literal is determined in a branch for the top query it can be stored in a table and its value used to speed up the computations of other branches within the same top query.

Goal-driven abductive reasoning is elegantly modelled by top-down abductive-query-solving. By taking a hypotheses assumption approach, enjoying Relevance, MH semantics caters well for this convenient problem representation and reasoning category.

Many applications have been developed using the Stable Model/Answer-set semantics as the underlying platform. These generally tend to be focused on solving problems that require complete knowledge, such as search problems where all the knowledge represented is relevant to the solutions. However, as Knowledge Bases increase in size and complexity, and as merging and updating of KBs becomes more and more common, e.g. for Semantic Web applications, [11], partial knowledge problem solving importance grows, as the need to ensure overall consistency of the merged/updated KBs.

The Minimal Hypotheses semantics is intended to, and can be used in *all* the applications where the Stable Models/Answer-Sets semantics are themselves used to model KRR and search problems, *plus* all applications where query answering (both under a credulous mode of reasoning and under a skeptical one) is intented, *plus* all applications where abductive reasoning is needed. The MH semantics aims to be a sound theoretical platform for 2-valued (possibly abductive) reasoning with logic programs.

Much work still remains to be done that can be rooted in this platform contribution. The general topics of using non-normal logic programs (allowing for negation, default and/or explicit, in the heads of rules) for Belief Revision, Updates, Preferences, etc., are *per se* orthogonal to the semantics issue, and therefore, all these subjects can now be addressed with Minimal Hypotheses semantics as the underlying platform.

Importantly, MH can guarantee the liveness of updated and self-updating LP programs such as those of EVOLP [1] and related applications. The Minimal Hypotheses semantics still has to be thoroughly compared with Revised Stable Models [15], PStable Models [14], and other related semantics.

In summary, we have provided a fresh platform on which to re-examine ever present issues in Logic Programming and its uses, which purports to provide a natural continuation and improvement of LP development.

References

1. J.J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca et al., editor, *Procs. JELIA'02*, volume 2424 of *LNCS*, pages 50–61. Springer, 2002.
2. S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001.
3. Gerhard Brewka. Logic programming with ordered disjunction. In *In Proceedings of AAAI-02*, pages 100–105. AAAI Press, 2002.
4. J. Dix. A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundam. Inform.*, 22(3):227–255, 1995.
5. J. Dix. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundam. Inform.*, 22(3):257–288, 1995.
6. Jürgen Dix, J Urgen Dix, Georg Gottlob, Wiktor Marek, and Cecylia Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996.
7. P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *AI*, 77(2):321–358, 1995.
8. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Procs. ICLP'88*, pages 1070–1080, 1988.
10. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren et al., editor, *ICLP*, pages 579–597. MIT Press, 1990.
11. A. S. Gomes, J. J. Alferes, and T. Swift. Implementing query answering for hybrid mknf knowledge bases. In M. Carro et al., editor, *PADL'10*, volume 5937 of *LNCS*, pages 25–39. Springer, 2010.
12. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.
13. R. C. Moore. Semantical considerations on nonmonotonic logic. *AI*, 25(1):75–94, 1985.
14. M. Osorio and J. C. Nieves. Pstable semantics for possibilistic logic programs. In *MICAI'07*, volume 4827 of *LNCS*, pages 294–304. Springer, 2007.
15. L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In C. Bento et al., editor, *Procs. EPIA'05*, volume 3808 of *LNAI*, pages 29–42. Springer, 2005.
16. L. M. Pereira and A. M. Pinto. Approved models for normal logic programs. In N. Dershowitz and A. Voronkov, editors, *Procs. LPAR'07*, volume 4790 of *LNAI*. Springer, 2007.
17. L. M. Pereira and A. M. Pinto. Reductio ad absurdum argumentation in normal logic programs. In G. Simari et al., editor, *ArgNMR'07-LPNMR'07*, pages 96–113. Springer, 2007.
18. L. M. Pereira and A. M. Pinto. *Collaborative vs. Conflicting Learning, Evolution and Argumentation, in: Oppositional Concepts in Computational Intelligence*. Studies in Computational Intelligence 155. Springer, 2008.
19. A. M. Pinto. *Every normal logic program has a 2-valued semantics: theory, extensions, applications, implementations*. PhD thesis, Universidade Nova de Lisboa, 2011.
20. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. on Computing*, 1(2):146–160, 1972.
21. C. Witteveen. Every normal program has a nearly stable model. In J. Dix, L.M. Pereira, and T.C. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming*, volume 927 of *Lecture Notes in Artificial Intelligence*, pages 68–84. Springer Verlag, Berlin, 1995.

Every Formula-Based Logic Program Has a Least Infinite-Valued Model

Rainer Lüdecke

Universität Tübingen, Wilhelm-Schickard-Institut,
Sand 13, 72076 Tübingen, Germany
luedecke@informatik.uni-tuebingen.de

Abstract. Every definite logic program has as its meaning a least Herbrand model with respect to the program-independent ordering $\subseteq$. In the case of normal logic programs there do not exist least models in general. However, according to a recent approach by Rondogiannis and Wadge, who consider infinite-valued models, every normal logic program does have a least model with respect to a program-independent ordering. We show that this approach can be extended to formula-based logic programs (i.e., finite sets of rules of the form $A \leftarrow \phi$ where A is an atom and ϕ an arbitrary first-order formula). We construct for a given program P an interpretation M_P and show that it is the least of all models of P.

Keywords: Logic programming, semantics of programs, negation-as-failure, infinite-valued logics, set theory

1 Introduction

It is well-known that every definite logic program P has a Herbrand model and the intersection of all its Herbrand models is also a model of P. We call it the least Herbrand model or the canonical model of P and constitute that it is the intended meaning of the program. If we consider a normal logic program P it is more complicated to state the intended meaning of the program because the intersection of all its models is not necessarily a model. There are many approaches to overcome that problem. The existing approaches are not purely model-theoretic (i.e., there are normal logic programs that have the same models but different intended meanings). However, there is a recent purely model-theoretic approach of P. Rondogiannis and W. Wadge [3]. They prove that every normal logic program has a least infinite-valued model. Their work is based on an infinite set of truth values, ordered as follows:

$$\mathcal{F}_0 < \mathcal{F}_1 < ... < \mathcal{F}_\alpha < ... < 0 < ... < \mathcal{T}_\alpha < ... < \mathcal{T}_1 < \mathcal{T}_0$$

Intuitively, $\mathcal{F}_0$ and $\mathcal{T}_0$ are the classical truth values *False* and *True*, 0 is the truth value *Undefined* and α is an arbitrary countable ordinal. The considered ordering of the interpretations is a program-independent ordering on the infinite-valued interpretations and generalizes the classical ordering on the Herbrand interpretations. The intended meaning of a normal logic program is, as in the classical case, stated as the unique minimal infinite-valued model of P. Furthermore, they show that the 3-valued interpretation that results from the least infinite-valued model of P by collapsing all true values to True and all false values to False coincides with the well-founded model of P introduced in [2].
Inspired by [4] we consider in this paper formula-based logic programs. A formula-based logic program is a finite set of rules of the form $A \leftarrow \phi$, where A is an atomic formula and ϕ is an arbitrary first-order formula. We show that the construction methods to obtain the least infinite-valued model of a normal logic program P given in [3] can be adapted to formula-based logic programs. The initial step to carry out this adaption is the proof of two extension theorems. Informally speaking, these theorems state that a complex formula shows the same behavior as an atomic formula. While Rondogiannis and Wadge [3] make use of the fact that the bodies of normal program rules are conjunctions of negative or positive atoms, we instead make use of one of the extension theorems. The second step to achieve the adaption is the set-theoretical fact that the least uncountable cardinal $\aleph_1$ is regular (i.e., the limit of a countable sequence of countable ordinals is in $\aleph_1$). Contrary to the bodies of normal program rules, the bodies of formula-based program

rules can refer a ground atom to a countably infinite set of ground atoms. This is the reason why we must use in our approach $\aleph_1$ many iteration steps in the construction of the least model of a given program P in conjunction with the regularity of $\aleph_1$. In [3] ω many iteration steps in conjunction with the fact that the limit of a finite sequence of natural numbers is once again a natural number is sufficient to construct the least model. Towards the end of the paper, we use again the regularity of $\aleph_1$ to show that there is a countable ordinal $\delta_{\max}$ with the property that every least model of a formula-based logic-program refers only to truth values of the form $\mathcal{T}_\alpha$ or $\mathcal{F}_\alpha$ or 0, where $\alpha < \delta_{\max}$. This implies that we only need a very small fragment of the truth values if we consider the meaning of a formula-based logic program. Finally, we show that the 3-valued interpretation that results from the least infinite-valued model of a given formula-based logic program by collapsing all true values to True and all false values to False, is a model of P in the sense of [2]. But compared to the case of normal logic programs, the collapsed least infinite-valued model of a formula-based logic program is not a minimal 3-valued model of P in general. However, there is a simple restriction for the class of formula-based programs such that the collapsed model is minimal in general.
At this point we would like to mention that we did not develop the theory presented in this paper with respect to applied logic. We have a predominantly theoretical interest in extending the notion of *inductive definition* to a wider class of rules.
We make heavy use of ordinal numbers in this paper. Therefore, we included an appendix with a short introduction to ordinal numbers for those readers who are not familiar with this part of set theory. Moreover, one can find the omitted proofs and a detailed discussion of an example within the appendix. It is downloadable at:
`http://www-ls.informatik.uni-tuebingen.de/luedecke/luedecke.html`

2 Infinite-Valued Models

We are interested in logic programs based on a first-order Language $\mathcal{L}$ with finitely many predicate symbols, function symbols, and constants.

Definition 1. The alphabet of $\mathcal{L}$ consists of the following symbols, where the numbers n, m, l, s_1,...,s_n, r_1,...,r_m are natural numbers such that $n, l, r_i \geq 1$ and $m, s_i \geq 0$ hold:

1. Predicate symbols: $P_1, ..., P_n$ with assigned arity $s_1, ..., s_n$
2. Function symbols: $f_1, ..., f_m$ with assigned arity $r_1, ..., r_m$
3. Constants (abbr.: Con): $c_1, ..., c_l$
4. Variables (abbr.: Var): x_k provided that $k \in \mathbb{N}$
5. Connectives: $\wedge, \vee, \neg, \forall, \exists, \bot, \top$
6. Punctuation symbols: '(', ')' and ','

The natural numbers $n, m, l, s_1, ..., s_n, r_1, ..., r_m \in \mathbb{N}$ are fixed and the language $\mathcal{L}$ only depends on these numbers. If we consider different languages of this type, we will write $\mathcal{L}_{n,m,l,(s_i),(r_i)}$ instead of $\mathcal{L}$ to prevent confusion. The following definitions depend on $\mathcal{L}$. However, to improve readability, we will not mention this again.

Definition 2. The set of *terms* Term is the smallest one satisfying:

1. Constants and variables are in Term.
2. If $t_1, ..., t_{r_i} \in$ Term, then $f_i(t_1, ..., t_{r_i}) \in$ Term.

Definition 3. The *Herbrand universe* H_U is the set of ground terms (i.e., terms that contain no variables).

Definition 4. The set of *formulas* Form is the smallest one satisfying:

1. $\bot$ and $\top$ are elements of Form.
2. If $t_1, ..., t_{s_k} \in$ Term, then $P_k(t_1, ..., t_{s_k}) \in$ Form.
3. If $\phi, \psi \in$ Form and $v \in$ Var, then $\neg(\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $\forall v(\phi)$, $\exists v(\phi) \in$ Form.

An *atom* is a formula only constructed by means of 1.) or 2.) and a *ground atom* is an atom that contains no variables.

Definition 5. The *Herbrand base* H_B is the set of all ground atoms, except $\bot$ and $\top$.

Definition 6. A *(formula-based) rule* is of the form $A \leftarrow \phi$ where ϕ is an arbitrary formula and A is an arbitrary atom provided that $A \neq \top$ and $A \neq \bot$.
A *(formula-based) logic program* is a finite set of (formula-based) rules. Notice that we write $A \leftarrow$ instead of $A \leftarrow \top$. Remember $A \leftarrow \phi$ is called a *normal rule* (resp. *definite rule*) if ϕ is a conjunction of literals (resp. positive literals). A finite set of normal (resp. definite) rules is a *normal (resp. definite) program*.

Definition 7. The set of truth values W is given by

$$W := \{\langle 0, n\rangle ;\ n \in \aleph_1\} \cup \{0\} \cup \{\langle 1, n\rangle ;\ n \in \aleph_1\}\,.$$

Additionally, we define a strict linear ordering $<$ on W as follows:

1. $\langle 0, n\rangle < 0$ and $0 < \langle 1, n\rangle$ for all $n \in \aleph_1$
2. $\langle w, x\rangle < \langle y, z\rangle$ iff
   $(w = 0 = y$ and $x \in z)$ or $(w = 1 = y$ and $z \in x)$ or $(w = 0$ and $y = 1)$

We define $\mathcal{F}_i := \langle 0, i\rangle$ and $\mathcal{T}_i := \langle 1, i\rangle$ for all $i \in \aleph_1$. $\mathcal{F}_i$ is a *false* value and $\mathcal{T}_i$ is a *true* value. The value 0 is the *undefined* value. The following summarizes the situation ($i \in \aleph_1$):

$$\mathcal{F}_0 < \mathcal{F}_1 < \mathcal{F}_2 < ... < \mathcal{F}_i < ... < 0 < ... < \mathcal{T}_i < ... < \mathcal{T}_2 < \mathcal{T}_1 < \mathcal{T}_0$$

Definition 8. The *degree* (abbr.: deg) of a truth value is given by $\deg(0) := \infty$, $\deg(\mathcal{F}_\alpha) := \alpha$, and $\deg(\mathcal{T}_\alpha) := \alpha$ for all $\alpha \in \aleph_1$.

Definition 9. An *(infinite-valued Herbrand) interpretation* I is a function from the Herbrand base H_B to the set of truth values W. A *variable assignment* h is a mapping from Var to H_U.

Definition 10. Let I be an interpretation and $w \in W$ be a truth value, then $I\|w$ is defined as the inverse image of w under I (i.e., $I\|w = \{A \in H_B;\ I(A) = w\}$).

Definition 11. Let I and J be interpretations and $\alpha \in \aleph_1$. We write $I =_\alpha J$, if for all $\beta \leq \alpha$, $I\|\mathcal{F}_\beta = J\|\mathcal{F}_\beta$ and $I\|\mathcal{T}_\beta = J\|\mathcal{T}_\beta$.

Definition 12. Let I and J be interpretations and $\alpha \in \aleph_1$. We write $I \sqsubseteq_\alpha J$, if for all $\beta < \alpha$, $I =_\beta J$ and furthermore $J\|\mathcal{F}_\alpha \subseteq I\|\mathcal{F}_\alpha$ & $I\|\mathcal{T}_\alpha \subseteq J\|\mathcal{T}_\alpha$. We write $I \sqsubset_\alpha J$, if $I \sqsubseteq_\alpha J$ and $I \neq_\alpha J$.

Now we define a partial ordering $\sqsubseteq_\infty$ on the set of all interpretations. It is easy to see that this ordering generalizes the classical partial ordering $\subseteq$ on the set of 2-valued Herbrand interpretations.

Definition 13. Let I and J be interpretations. We write $I \sqsubset_\infty J$, if there exists an $\alpha \in \aleph_1$ such that $I \sqsubset_\alpha J$. We write $I \sqsubseteq_\infty J$, if $I \sqsubset_\infty J$ or $I = J$.

Remark 1. To motivate these definitions let us briefly recall the classical 2-valued situation. Therefore let us pick two (2-valued) Herbrand interpretations $I, J \subseteq H_B$. Considering these, it becomes apparent that $I \subseteq J$ holds if and only if the set of ground atoms that are false w.r.t. J is a subset of the set of ground atoms that are false w.r.t. I and the set of ground atoms that are true w.r.t. I is a subset of the set of ground atoms that are true w.r.t. J.

Definition 14. Let h be a variable assignment. The *semantics of terms* is given by (with respect to h):

1. $[\![c]\!]_h = c$ if c is a constant.
2. $[\![v]\!]_h = h(v)$ if v is a variable.
3. $[\![f_i(t_1, ..., t_{r_i})]\!]_h = f_i([\![t_1]\!]_h, ..., [\![t_{r_i}]\!]_h)$ if $1 \leq i \leq m$ and $t_1, ..., t_{r_i} \in$ Term.

Before we start to talk about the semantics of formulas, we have to show that every subset of W has a *least upper bound* (abbr: sup) and a *greatest lower bound* (abbr: inf). The proof of the following lemma is left to the reader. The proof is using the fact that every nonempty subset of $\aleph_1$ has a least element.

Lemma 1. *For every subset $M \subseteq W$ the least upper bound* $\sup M$ *and the greatest lower bound* $\inf M$ *exist in W. Moreover,* $\sup M \in \{\mathcal{T}_\alpha;\ \alpha \in \aleph_1\}$ *implies that* $\sup M \in M$ *and on the other hand* $\inf M \in \{\mathcal{F}_\alpha;\ \alpha \in \aleph_1\}$ *implies that* $\inf M \in M$.

Definition 15. Let I be an interpretation and h be a variable assignment. The *semantics of formulas* is given by (with respect to I and h):

1. If $t_1, ..., t_{s_k} \in$ Term, then $[\![P_k(t_1, ..., t_{s_k})]\!]_h^I = I\,(P_k([\![t_1]\!]_h, ..., [\![t_{s_k}]\!]_h))$. Additionally, the semantics of $\top$ and $\bot$ is given by $[\![\top]\!]_h^I = \mathcal{T}_0$ and $[\![\bot]\!]_h^I = \mathcal{F}_0$.
2. If $\phi, \psi \in$ Form and v an arbitrary variable, then

$$[\![\phi \wedge \psi]\!]_h^I = \min\{[\![\phi]\!]_h^I, [\![\psi]\!]_h^I\},$$

$$[\![\phi \vee \psi]\!]_h^I = \max\{\ [\![\phi]\!]_h^I\ ,\ [\![\psi]\!]_h^I\ \},$$

$$[\![\exists v(\phi)]\!]_h^I = \sup\{[\![\phi]\!]_{h[v \mapsto u]}^I;\ u \in H_U\},$$

$$[\![\forall v(\phi)]\!]_h^I = \inf\{[\![\phi]\!]_{h[v \mapsto u]}^I;\ u \in H_U\}$$

and

$$[\![\neg(\phi)]\!]_h^I = \begin{cases} \mathcal{T}_{\alpha+1}, & \text{if } [\![\phi]\!]_h^I = \mathcal{F}_\alpha \\ \mathcal{F}_{\alpha+1}, & \text{if } [\![\phi]\!]_h^I = \mathcal{T}_\alpha \\ 0, & \text{otherwise} \end{cases}.$$

Definition 16. Let $A \leftarrow \phi$ be a rule, P a program and I an interpretation. Then I *satisfies* $A \leftarrow \phi$ if for all variable assignment h the property $[\![A]\!]_h^I \geq [\![\phi]\!]_h^I$ holds. Furthermore, I is a *model* of P if I satisfies all rules of P.

Definition 17. Let $A \leftarrow \phi$ be a rule and σ be a variable substitution (i.e., a function from Var to Term with finite support). Then, $A\sigma \leftarrow \phi\sigma$ is a *ground instance* of the rule $A \leftarrow \phi$ if $A\sigma \in H_B$ and all variables in $\phi\sigma$ are in the scope of a quantifier. It is easy to see that that $[\![A\sigma]\!]_h^I$ and $[\![\phi\sigma]\!]_h^I$ (with respect to an interpretation I and a variable assignment h) depend only on I. That is why we write also $[\![A\sigma]\!]^I$ and $[\![\phi\sigma]\!]^I$. We denote the *set of all ground instances* of a program P with P_G.

Example 1. Consider the formula-based program P given by the set of rules

$$\{P(c) \leftarrow,\ R(x) \leftarrow \neg P(x),\ P(Sx) \leftarrow \neg R(x),\ Q \leftarrow \forall x\,(P(x))\}.$$

Then it is easy to prove that the Herbrand interpretation $I = \{P(S^n c) \mapsto \mathcal{T}_{2n};\ n \in \mathbb{N}\} \cup \{R(S^n c) \mapsto \mathcal{F}_{2n+1};\ n \in \mathbb{N}\} \cup \{Q \mapsto \mathcal{T}_\omega\}$ is a model of P. Moreover, using the results of this paper one can show that it is also the least Herbrand model of P.

Remark 2. Before we proceed we want to give a short informal but intuitive description of the semantics given above. Let us consider two rabbits named Bugs Bunny and Roger Rabbit. We know about them, that Bugs Bunny is a grey rabbit and if Roger Rabbit is not a grey rabbit, then he is a white one. This information can be understood as a normal logic program:

grey(Bugs Bunny) $\Leftarrow$

white(Roger Rabbit) $\Leftarrow$ *not grey(Roger Rabbit)*

There is no doubt that *Bugs Bunny is grey* is true because it is a fact. There is also no doubt that every try to prove that *Roger Rabbit is grey* will fail. Hence, using the negation-as-failure rule, we can infer that *Roger Rabbit is white* is also true. But everybody would agree that there is a difference of quality between the two statements because negation-as-failure is not a sound inference rule. The approach of [3] suggests that the ground atom *grey(Bugs Bunny)* receives the best possible truth value named $\mathcal{T}_0$ because it is a fact of the program. The atom *grey(Roger Rabbit)* receives the worst possible truth value named $\mathcal{F}_0$ because of the negation-as-failure approach. Hence, using the above semantics for negation, *white(Roger Rabbit)* receives only the second best truth value $\mathcal{T}_1$.

3 The Immediate Consequence Operator

Definition 18. Let P be a program, then the *immediate consequence operator* T_P for the program P is a mapping from and into $\{I;\ I \text{ is an interpretation}\}$, where $T_P(I)$ maps an $A \in H_B$ to $T_P(I)(A) := \sup\{[\![\phi]\!]^I;\ A \leftarrow \phi \in P_G\}$. (Notice that P_G can be infinite and hence we cannot use max instead of sup.)

Definition 19. Let α be an arbitrary countable ordinal. A function T from and into the set of interpretations is called *α-monotonic* iff for all interpretations I and J the property $I \sqsubseteq_\alpha J \Rightarrow T(I) \sqsubseteq_\alpha T(J)$ holds.

We will show that T_P is α-monotonic. Before we will give the proof of this property, we have to prove the first extension theorem.

Theorem 1 (Extension Theorem I). *Let α be an arbitrary countable ordinal and I, J two interpretations provided that $I \sqsubseteq_\alpha J$. The following properties hold for every formula ϕ:*

1. *If $\mathcal{F}_0 \leq w \leq \mathcal{F}_\alpha$ and h an assignment, then $[\![\phi]\!]^J_h = w \ \Rightarrow\ [\![\phi]\!]^I_h = w$.*
2. *If $\mathcal{T}_\alpha \leq w \leq \mathcal{T}_0$ and h an assignment, then $[\![\phi]\!]^I_h = w \ \Rightarrow\ [\![\phi]\!]^J_h = w$.*
3. *If $\deg(w) < \alpha$ and h an assignment, then $[\![\phi]\!]^I_h = w \ \Leftrightarrow\ [\![\phi]\!]^J_h = w$.*

Proof. We show these statements by induction on ϕ. Let $I_H(X)$ be an abbreviation for 1. and 2. and 3., where ϕ is replaced by X (induction hypothesis).
Case 1: $\phi = \top$ or $\phi = \bot$. In this case 1., 2., and 3. are obviously true.
Case 2: $\phi = P_k(t_1, ..., t_{s_k})$. 1., 2., and 3. follow directly from $I \sqsubseteq_\alpha J$.
Case 3: $\phi = \neg(A)$. We assume that $I_H(A)$. We show simultaneously that 1., 2. and 3. also hold. Therefore, we choose an assignment h and a truth value w such that $\mathcal{F}_0 \leq w \leq \mathcal{F}_\alpha$ resp. $\mathcal{T}_\alpha \leq w \leq \mathcal{T}_0$ resp. $\deg(w) < \alpha$. Assume that $[\![\phi]\!]^J_h = w$ resp. $[\![\phi]\!]^I_h = w$ resp. $[\![\phi]\!]^{K_1}_h = w$ (where $K_1 = I$ and $K_2 = J$ or $K_1 = J$ and $K_2 = I$). Using Definition 15 we get that $\mathcal{T}_{\alpha-1} \leq [\![A]\!]^J_h \leq \mathcal{T}_0$ resp. $\mathcal{F}_0 \leq [\![A]\!]^I_h \leq \mathcal{F}_{\alpha-1}$ resp. $\deg([\![A]\!]^{K_1}_h) < \alpha - 1$. Then, from the third part of $I_H(A)$, $[\![A]\!]^J_h = [\![A]\!]^I_h$ resp. $[\![A]\!]^I_h = [\![A]\!]^J_h$ resp. $[\![A]\!]^{K_1}_h = [\![A]\!]^{K_2}_h$. Finally, using Definition 15, we get that $[\![\phi]\!]^I_h = w$ resp. $[\![\phi]\!]^J_h = w$ resp. $[\![\phi]\!]^{K_2}_h = w$.

Before we can go on with the next case, we must prove the following technical lemma.

Lemma 2. *We use the same assumptions as in Theorem 1. Let $\mathcal{I}$ be a set of indices, A_i ($i \in \mathcal{I}$) a formula provided that $I_H(A_i)$ and h_i ($i \in \mathcal{I}$) an assignment. We define* $\inf_K := \inf\{[\![A_i]\!]^K_{h_i}; i \in \mathcal{I}\}$ *and* $\sup_K := \sup\{[\![A_i]\!]^K_{h_i}; i \in \mathcal{I}\}$ *(where $K = I, J$). Then the following holds:*

1. $\inf_J = \mathcal{F}_\gamma \ \Rightarrow\ \inf_I = \mathcal{F}_\gamma$ *(for all $\gamma \leq \alpha$)*
2. $\inf_I = \mathcal{T}_\gamma \ \Rightarrow\ \inf_J = \mathcal{T}_\gamma$ *(for all $\gamma \leq \alpha$)*
3. $\inf_I = w \ \Leftrightarrow\ \inf_J = w$ *(for all w provided that $\deg(w) < \alpha$)*
4. $\sup_J = \mathcal{F}_\gamma \ \Rightarrow\ \sup_I = \mathcal{F}_\gamma$ *(for all $\gamma \leq \alpha$)*
5. $\sup_I = \mathcal{T}_\gamma \ \Rightarrow\ \sup_J = \mathcal{T}_\gamma$ *(for all $\gamma \leq \alpha$)*
6. $\sup_I = w \ \Leftrightarrow\ \sup_J = w$ *(for all w provided that $\deg(w) < \alpha$)*

Proof. 1.: Assume that $\inf_J = \mathcal{F}_\gamma$. Using Lemma 1 we get that there exists an i_0 such that $[\![A_{i_0}]\!]^J_{h_{i_0}} = \mathcal{F}_\gamma$. Then, from the first part of $I_H(A_{i_0})$, $[\![A_{i_0}]\!]^I_{h_{i_0}} = \mathcal{F}_\gamma$. This implies that $[\![A_{i_0}]\!]^I_{h_{i_0}} \leq [\![A_i]\!]^I_{h_i}$ for all $i \in \mathcal{I}$. (Since otherwise we had that there exists a $j_0 \in \mathcal{I}$ such that $[\![A_{j_0}]\!]^I_{h_{j_0}} < \mathcal{F}_\gamma$. Then, using the third part of $I_H(A_{j_0})$, it would also be $[\![A_{j_0}]\!]^J_{h_{j_0}} < \mathcal{F}_\gamma$. But this contradicts our assumption $\inf_J = \mathcal{F}_\gamma$.) Finally, we get that $\inf_I = \mathcal{F}_\gamma$.
2.: Assume now, that $\inf_I = \mathcal{T}_\gamma$. Then $\mathcal{T}_\gamma \leq [\![A_i]\!]^I_{h_i}$ for all $i \in \mathcal{I}$. Using part two of $I_H(A_i)$, we get that $[\![A_i]\!]^I_{h_i} = [\![A_i]\!]^J_{h_i}$ for all i. This implies $\inf_J = \mathcal{T}_\gamma$.
3.: Due to 1. and 2., it only remains to show $(\inf_J = \mathcal{T}_\gamma \ \Rightarrow\ \inf_I = \mathcal{T}_\gamma)$ and $(\inf_I = \mathcal{F}_\gamma \ \Rightarrow\ \inf_J = \mathcal{F}_\gamma)$ for $\gamma < \alpha$. Assume that $\inf_J = \mathcal{T}_\gamma$ (where $\gamma < \alpha$). Then $\mathcal{T}_\gamma \leq [\![A_i]\!]^J_{h_i}$ for all $i \in \mathcal{I}$ and this implies, using the third part of $I_H(A_i)$, $[\![A_i]\!]^J_{h_i} = [\![A_i]\!]^I_{h_i}$ for all i. Finally, we get that $\inf_I = \mathcal{T}_\gamma$.
For the latter case assume that $\inf_I = \mathcal{F}_\gamma$ ($\gamma < \alpha$). Then there exists an i_0 such that $[\![A_{i_0}]\!]^I_{h_{i_0}} = \mathcal{F}_\gamma$ (Lemma 1). Then, using the third part of $I_H(A_{i_0})$, we get that $[\![A_{i_0}]\!]^J_{h_{i_0}} = \mathcal{F}_\gamma$. This implies that $[\![A_{i_0}]\!]^J_{h_{i_0}} \leq [\![A_i]\!]^J_{h_i}$

for all $i \in \mathcal{I}$. (Since otherwise we had that there exists a $j_0 \in \mathcal{I}$ such that $[\![A_{j_0}]\!]^I_{h_{j_0}} < \mathcal{F}_\gamma$, see proof of statement 1.) Finally, we get that $\inf_J = \mathcal{F}_\gamma$.
We will not give the proofs of 4., 5., and 6. here, because they are similar to 1., 2., and 3.. □

Case 4: $\phi = A \wedge B$. Assume that $I_H(A)$ and $I_H(B)$. Let h be an arbitrary assumption. We define $\mathcal{I} := \{1, 2\}$, $h_1 := h$, $h_2 := h$, $A_1 := A$ and $A_2 := B$. Then $I_H(A_i)$ for $i = 1, 2$, $[\![\phi]\!]^J_h = \min\{[\![A]\!]^J_h, [\![B]\!]^J_h\} = \inf_J$ and $[\![\phi]\!]^I_h = \min\{[\![A]\!]^I_h, [\![B]\!]^I_h\} = \inf_I$. Then, using 1., 2. and 3. of Lemma 2, we get that 1., 2. and 3. of Theorem 1 hold.
Case 5: $\phi = A \vee B$. Replace min by max and inf by sup in the proof above and use 4., 5. and 6. of Lemma 2 instead of 1., 2. and 3..
Case 6: $\phi = \forall v(A)$. Assume that $I_H(A)$ and let h be an arbitrary assumption.
We define $\mathcal{I} := \{u;\ u \in H_U\}$, $h_u := h[v \mapsto u]$ and $A_u := A$ for all $u \in H_U$. Then $I_H(A_u)$ for all $u \in \mathcal{I}$, $[\![\phi]\!]^J_h = \inf\{[\![A]\!]^J_{h[v\mapsto u]};\ u \in H_U\} = \inf_J$, and $[\![\phi]\!]^I_h = \inf\{[\![A]\!]^I_{h[v\mapsto u]};\ u \in H_U\} = \inf_I$. Then, using 1., 2. and 3. of Lemma 2, we get that 1., 2. and 3. of Theorem 1 hold.
Case 7: $\phi = \exists v(A)$. Replace inf by sup in the proof above and use 4., 5. and 6. of Lemma 2 instead of 1., 2. and 3.. □

Lemma 3. *The immediate consequence operator T_P of a given program P is α-monotonic for all countable ordinals α.*

Proof. The proof is by transfinite induction on α. Assume the lemma holds for all $\beta < \alpha$ (induction hypothesis). We demonstrate that it also holds for α. Let I, J be two interpretations such that $I \sqsubseteq_\alpha J$. Then, using the induction hypothesis, we get that

$$T_P(I) =_\beta T_P(J) \text{ for all } \beta < \alpha. \tag{1}$$

It remains to show that $T_P(I) \parallel \mathcal{T}_\alpha \subseteq T_P(J) \parallel \mathcal{T}_\alpha$ and that $T_P(J) \parallel \mathcal{F}_\alpha \subseteq T_P(I) \parallel \mathcal{F}_\alpha$. For the first statement assume that $T_P(I)(A) = \mathcal{T}_\alpha$ for some $A \in H_B$. Then, using Lemma 1, there exists a ground instance $A \leftarrow \phi$ of P such that $[\![\phi]\!]^I = \mathcal{T}_\alpha$. But then, by Theorem 1, $[\![\phi]\!]^J = \mathcal{T}_\alpha$. This implies $\mathcal{T}_\alpha \leq T_P(J)(A)$. But this implies $\mathcal{T}_\alpha = T_P(J)(A)$. (Since $\mathcal{T}_\alpha < T_P(J)(A)$, using (1), would imply $\mathcal{T}_\alpha < T_P(I)(A)$.) For the latter statement assume that $T_P(J)(A) = \mathcal{F}_\alpha$ for some $A \in H_B$. This implies that $[\![\phi]\!]^J \leq \mathcal{F}_\alpha$ for every ground instance $A \leftarrow \phi$ of P. But then, using again Theorem 1, we get that $[\![\phi]\!]^I = [\![\phi]\!]^J$ for every ground instance $A \rightarrow \phi$ of P. Finally, this implies also $T_P(I)(A) = \mathcal{F}_\alpha$. □

Remark 3. The immediate consequence operator T_P is not monotonic with respect to $\sqsubseteq_\infty$. Consider the program $P = \{A \leftarrow \neg A\}$ and the interpretations I_1 and I_2 given by $I_1 := \{A \mapsto \mathcal{F}_0\}$ and $I_2 := \{A \mapsto 0\}$. Obviously, $I_1 \sqsubset_0 I_2$ and hence $I_1 \sqsubseteq_\infty I_2$. Using Definition 18, we get that $T_P(I_1) = sup\{[\![\neg A]\!]^{I_1}\} = \mathcal{T}_1$ and $T_P(I_2) = sup\{[\![\neg A]\!]^{I_2}\} = 0$. This implies $T_P(I_2) \sqsubset_1 T_P(I_1)$ (i.e., $T_P(I_1) \sqsubseteq_\infty T_P(I_2)$ does not hold).

4 Construction of the Minimum Model

In this section we show how to construct the interpretation M_P of a given formula-based logic program P. We will give the proof that M_P is a model of P and that it is the least of all models of P in the next section. In [3] the authors give a clear informal description of the following construction:
"As a first approximation to M_P, we start (...) iterating the T_P on $\emptyset$ until both the set of atoms that have a $\mathcal{F}_0$ value and the set of atoms having $\mathcal{T}_0$ value, stabilize. We keep all these atoms whose values have stabilized and reset the values of all remaining atoms to the next false value (namely $\mathcal{F}_1$). The procedure is repeated until the $\mathcal{F}_1$ and $\mathcal{T}_1$ values stabilize, and we reset the remaining atoms to a value equal to $\mathcal{F}_2$, and so on. Since the Herbrand Base of P is countable, there exists a countable ordinal δ for which this process will not produce any new atoms having $\mathcal{F}_\delta$ or $\mathcal{T}_\delta$ values. At this point we stop iteration and reset all remaining atoms to the value 0."

Definition 20. Let P be a program, I an interpretation, and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. We define by recursion on the ordinal $\beta \in \Omega$ the interpretation $T^\beta_{P,\alpha}(I)$ as follows:

$T^0_{P,\alpha}(I) := I$ and if β is a successor ordinal, then $T^\beta_{P,\alpha} := T_P(T^{\beta-1}_{P,\alpha})$. If $0 < \beta$ is a limit ordinal and $A \in H_B$, then

$$T^\beta_{P,\alpha}(I)(A) := \begin{cases} I(A), & \text{if } \deg(I(A)) < \alpha \\ \mathcal{T}_\alpha, & \text{if } A \in \bigcup_{\gamma\in\beta} T^\gamma_{P,\alpha}(I) \| \mathcal{T}_\alpha \\ \mathcal{F}_\alpha, & \text{if } A \in \bigcap_{\gamma\in\beta} T^\gamma_{P,\alpha}(I) \| \mathcal{F}_\alpha \\ \mathcal{F}_{\alpha+1}, & \text{otherwise} \end{cases}.$$

Lemma 4. *Let P be a program, I an interpretation and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. Then the following holds:*

1. *For all limit ordinals $0 < \gamma \in \Omega$ and all interpretations M the condition $\forall \beta < \gamma : T^\beta_{P,\alpha}(I) \sqsubseteq_\alpha M$ implies $T^\gamma_{P,\alpha}(I) \sqsubseteq_\alpha M$.*
2. *For all $\beta \leq \gamma \in \Omega$ the property $T^\beta_{P,\alpha}(I) \sqsubseteq_\alpha T^\gamma_{P,\alpha}(I)$ holds.*

Proof. **1.**: The proof follows directly from the above definition.
2.: One can prove the second statement with induction, using the assumption $I \sqsubseteq_\alpha T_P(I)$, the fact that T_P is α-monotonic, the fact that $\sqsubseteq_\alpha$ is transitive and at limit stage the first statement of this lemma. □

At this point, we have to consider a theorem of Zermelo-Fraenkel axiomatic set theory with the Axiom of Choice (ZFC). In the case of normal logic programs this theorem is not necessary, because in the bodies of normal logic programs do not appear "∀" or "∃". One can find the proof of the theorem in [1].

Definition 21. Let $\alpha > 0$ be a limit ordinal. We say that an increasing β-sequence $(\alpha_\zeta)_{\zeta<\beta}$, β limit ordinal, is *cofinal* in α if $\sup\{\alpha_\zeta; \zeta < \beta\} = \alpha$. Similarly, $A \subseteq \alpha$ is *cofinal* in α if $\sup A = \alpha$. If α is an infinite limit ordinal, the *cofinality* of α is $cf(\alpha)$ = "the least limit ordinal β such that there is an increasing β-sequence $(\alpha_\zeta)_{\zeta<\beta}$ with $\sup\{\alpha_\zeta; \zeta < \beta\} = \alpha$". An infinite cardinal $\aleph_\alpha$ is *regular* if $cf(\aleph_\alpha) = \aleph_\alpha$.

Theorem 2. *Every cardinal of the form $\aleph_{\alpha+1}$ is regular. Particularly, $\aleph_1$ is regular.*

Theorem 3 (Extension Theorem II). *Let P be a program, I an interpretation, and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. Then for every formula $\phi \in$ Form and every assignment h the following hold:*

1. $[\![\phi]\!]_h^{T^{\aleph_1}_{P,\alpha}(I)} = [\![\phi]\!]_h^I$, *if* $\deg([\![\phi]\!]_h^I) < \alpha$ (C1)
2. $[\![\phi]\!]_h^{T^{\aleph_1}_{P,\alpha}(I)} = \mathcal{T}_\alpha$, *if* $[\![\phi]\!]_h^{T^i_{P,\alpha}(I)} = \mathcal{T}_\alpha$ *for some* $i \in \aleph_1$ (C2)
3. $[\![\phi]\!]_h^{T^{\aleph_1}_{P,\alpha}(I)} = \mathcal{F}_\alpha$, *if* $[\![\phi]\!]_h^{T^i_{P,\alpha}(I)} = \mathcal{F}_\alpha$ *for all* $i \in \aleph_1$ (C3)
4. $\mathcal{F}_\alpha < [\![\phi]\!]_h^{T^{\aleph_1}_{P,\alpha}(I)} < \mathcal{T}_\alpha$ $\Leftrightarrow$ *not*(C1) *and not*(C2) *and not*(C3)

Proof. 1. and 2.: We get this using Lemma 4 and Theorem 1.
3.: We show this by induction on ϕ. We define $I_i := T^i_{P,\alpha}(I)$ and $I_\infty := T^{\aleph_1}_{P,\alpha}(I)$. Moreover, we use $I_H(X)$ as an abbreviation for

"for all assignments g the property $\forall i \in \aleph_1([\![X]\!]_g^{I_i} = \mathcal{F}_\alpha) \Rightarrow [\![X]\!]_g^{I_\infty} = \mathcal{F}_\alpha$ holds".

Case 1: $\phi = P_k(t_1, ..., t_{s_k})$ or $= \top, \bot$. This follows directly from Definition 20 respectively from Definition 15.
Case 2: $\phi = \neg(A)$. Assuming $\forall i \in \aleph_1$: $[\![\phi]\!]_h^{I_i} = \mathcal{F}_\alpha$ we conclude $\forall i \in \aleph_1 : [\![A]\!]_h^{I_i} = \mathcal{T}_{\alpha-1}$. Then, by Theorem 1, we get $[\![A]\!]_h^{I_\infty} = \mathcal{T}_{\alpha-1}$ and this implies $[\![\phi]\!]_h^{I_\infty} = \mathcal{F}_\alpha$.
Case 3: $\phi = A \wedge B$ or $\phi = A \vee B$. The following cases are more general than this case. Therefore, we will not give a proof here.
Case 4: $\phi = \exists v(A)$. We assume that $I_H(A)$ and for every $i \in \aleph_1$ we assume that $[\![\phi]\!]_h^{I_i} = \mathcal{F}_\alpha$. This implies $\sup\{[\![A]\!]^I_{h[v\mapsto u]};\ u \in H_U\} = \mathcal{F}_\alpha$ as well as $\forall i \in \aleph_1 \forall u \in H_U : [\![A]\!]^{I_i}_{h[v\mapsto u]} \leq \mathcal{F}_\alpha$. Now we show by case distinction that $\forall u \in H_U : [\![A]\!]^{I_\infty}_{h[v\mapsto u]} = [\![A]\!]^I_{h[v\mapsto u]}$ and this obviously implies $[\![\phi]\!]_h^{I_\infty} = \mathcal{F}_\alpha$. First we consider the case $[\![A]\!]^I_{h[v\mapsto u]} < \mathcal{F}_\alpha$. Then, using Lemma 4 and Theorem 1, we get that $[\![A]\!]^{I_\infty}_{h[v\mapsto u]} = [\![A]\!]^I_{h[v\mapsto u]}$. At least, we consider the other case $[\![A]\!]^I_{h[v\mapsto u]} = \mathcal{F}_\alpha$. We know that $\forall i \in \aleph_1 : [\![A]\!]^{I_i}_{h[v\mapsto u]} \leq \mathcal{F}_\alpha$.

But this implies $\forall i \in \aleph_1 : [\![A]\!]^{I_i}_{h[v\mapsto u]} = \mathcal{F}_\alpha$, since $\exists i \in \aleph_1 : [\![A]\!]^{I_i}_{h[v\mapsto u]} < \mathcal{F}_\alpha$ would imply (using Lemma 4 and Theorem 1) $[\![A]\!]^{I}_{h[v\mapsto u]} < \mathcal{F}_\alpha$, which is a contradiction. Finally, we get, by $I_H(A)$, that $[\![A]\!]^{I_\infty}_h = \mathcal{F}_\alpha = [\![A]\!]^{I}_{h[v\mapsto u]}$.
Case 5: $\phi = \forall v(A)$. We assume that $I_H(A)$ and for every $i \in \aleph_1$ we assume that $[\![\phi]\!]^{I_i}_h = \mathcal{F}_\alpha$. Then $\forall i \in \aleph_1 : \inf\{[\![A]\!]^{I_i}_{h[v\mapsto u]};\ u \in H_U\} = \mathcal{F}_\alpha$. This implies, using Lemma 1, $\forall i \in \aleph_1 \exists u \in H_U : [\![A]\!]^{I_i}_{h[v\mapsto u]} = \mathcal{F}_\alpha$. Next we choose for every $i \in \aleph_1$ an atom $u_i \in H_U$ with $[\![A]\!]^{I_i}_{h[v\mapsto u_i]} = \mathcal{F}_\alpha$ (Remark: We do not need the Axiom of Choice because H_U is countable). Then, using Lemma 4 and Theorem 1, $\forall i \in \aleph_1 \forall j \leq i \in \aleph_1 : [\![A]\!]^{I_j}_{h[v\mapsto u_i]} = \mathcal{F}_\alpha$. This implies that the mapping

$$\zeta : \{u_i; i \in \aleph_1\} \to \aleph_1 \cup \{\aleph_1\} : u \mapsto \begin{cases} \min\{j \in \aleph_1; [\![A]\!]^{I_j}_{h[v\mapsto u]} \neq \mathcal{F}_\alpha\}, & \text{if min exists} \\ \aleph_1, & \text{otherwise} \end{cases}$$

has the properties $\forall i \in \aleph_1 : \zeta(u_i) > i$ and $\sup\{\zeta(u_i); i \in \aleph_1\} = \aleph_1$. We assume now that $\forall u \in H_U \exists j \in \aleph_1 : [\![A]\!]^{I_j}_{h[v\mapsto u]} \neq \mathcal{F}_\alpha$. Then $\zeta(\{u_i; i \in \aleph_1\})$ is a countable subset of $\aleph_1$ and moreover cofinal in $\aleph_1$. But this is a contradiction to Theorem 2. Therefore we know that there exists an atom $u^* \in H_U$ such that $\forall i \in \aleph_1 : [\![A]\!]^{I_i}_{h[v\mapsto u^*]} = \mathcal{F}_\alpha$. Then, using $I_H(A)$, we get that $[\![A]\!]^{I_\infty}_{h[v\mapsto u^*]} = \mathcal{F}_\alpha$. This implies $[\![\phi]\!]^{I_\infty}_h \leq \mathcal{F}_\alpha$ and finally, using $[\![\phi]\!]^{I}_h = \mathcal{F}_\alpha$, Lemma 4 and Theorem 1, we get that $[\![\phi]\!]^{I_\infty}_h = \mathcal{F}_\alpha$.

4.:"$\Rightarrow$": We prove this by the method of contrapositive. We assume that (C1) or (C2) or (C3). Then, using 1., 2., and 3., we get that not($\mathcal{F}_\alpha < [\![\phi]\!]^{I_\infty}_h < \mathcal{T}_\alpha$) holds.
"$\Leftarrow$": We shall first consider the following Lemma.

Lemma 5. *Under the same conditions as in Theorem 3 for every formula* $\phi \in$ Form *and every assignment* h *the following hold:*

$$[\![\phi]\!]^{I_\infty}_h = \mathcal{T}_\alpha \quad \Rightarrow \quad [\![\phi]\!]^{I_i}_h = \mathcal{T}_\alpha \text{ for some } i \in \aleph_1$$

Proof. This proof is similar to the proof of Theorem 3 statement 3. (see Appendix). □

We prove "$\Leftarrow$" also by the method of contrapositive. We assume that $\mathcal{F}_\alpha < [\![\phi]\!]^{I_\infty}_h < \mathcal{T}_\alpha$ does not hold. We consider the three possible cases $\deg([\![\phi]\!]^{I_\infty}_h) < \alpha$, $[\![\phi]\!]^{I_\infty}_h = \mathcal{F}_\alpha$, and $[\![\phi]\!]^{I_\infty}_h = \mathcal{T}_\alpha$. Let us consider the first case (resp. the second case). Then, using Lemma 4 and Theorem 1, (C1) (resp. (C3)) holds. Now, we consider the latter case. Using Lemma 5 we get that (C2) holds. Finally, in every case (C1) or (C2) or (C3) holds. □

Definition 22. Let α be a countable ordinal and for every $\gamma < \alpha$ let I_γ be an interpretation such that $\forall \zeta \leq \gamma : I_\zeta =_\zeta I_\gamma$. Then the union of the interpretations I_γ ($\gamma < \alpha$) is a well-defined interpretation and given by the following definition:

$$\bigsqcup_{\gamma<\alpha} I_\gamma\ (A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \alpha \ \&\ I_\zeta(A) = \mathcal{F}_\zeta \\ \mathcal{T}_\zeta, & \text{if } \zeta < \alpha \ \&\ I_\zeta(A) = \mathcal{T}_\zeta \qquad (A \in H_B) \\ \mathcal{F}_\alpha, & \text{otherwise} \end{cases}$$

Remark 4. Using $\forall \zeta \leq \gamma : I_\zeta =_\zeta I_\gamma$ it is easy to prove that the union $\bigsqcup_{\gamma<\alpha} I_\gamma$ is a well-defined interpretation. Particularly if $\alpha = 0$, then the union is equal to the interpretation that maps all atoms of H_B to the truth value $\mathcal{F}_0$. This interpretation is sometimes denoted by $\emptyset$.

Lemma 6. *Let* P *be a program,* α *be a countable ordinal and for all* $\gamma < \alpha$ *an interpretation* I_γ *is given such that* $\forall \zeta < \gamma : I_\zeta =_\zeta I_\gamma$. *Then the following holds:*

$$\forall \gamma < \alpha\, (I_\gamma \sqsubseteq_{\gamma+1} T_P(I_\gamma)) \ \Rightarrow\ \bigsqcup_{\gamma<\alpha} I_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$$

Proof. We assume that

$$\forall\gamma < \alpha\,(I_\gamma \sqsubseteq_{\gamma+1} T_P(I_\gamma))\,. \tag{2}$$

First, we prove that $\forall\beta < \alpha :\ \bigsqcup_{\gamma<\alpha} I_\gamma =_\beta T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$. For all $\beta < \alpha$ we know that $I_\beta =_\beta \bigsqcup_{\gamma<\alpha} I_\gamma$. Then, using Lemma 3, $\forall\beta < \alpha : T_P(I_\beta) =_\beta T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$. This implies for all $\beta < \alpha$ the property $\bigsqcup_{\gamma<\alpha} I_\gamma =_\beta I_\beta =^{(2)}_\beta T_P(I_\beta) =_\beta T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$. We know that $\bigsqcup_{\gamma<\alpha} I_\gamma$ does not map to truth values w such that $\mathcal{F}_\alpha < w \leq \mathcal{T}_\alpha$. And this obviously implies $\bigsqcup_{\gamma<\alpha} I_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$. □

Lemma 7. *Let P be a program, α a countable ordinal, and I an interpretation. Then the following holds:*

$$I \sqsubseteq_\alpha T_P(I) \ \Rightarrow\ T^{\aleph_1}_{P,\alpha}(I) \sqsubseteq_{\alpha+1} T_P(T^{\aleph_1}_{P,\alpha}(I))$$

Proof. Again, we define $I_i := T^i_{P,\alpha}(I)$, $I_\infty := T^{\aleph_1}_{P,\alpha}(I)$. Let us assume that $I \sqsubseteq_\alpha T_P(I)$. First we prove $I_\infty \sqsubseteq_\alpha T_P(I_\infty)$. Using Lemma 4 we get that $\forall\gamma < \aleph_1 : I_\gamma \sqsubseteq_\alpha I_\infty$. Then, using Lemma 3, $\gamma \in \aleph_1 : I_{\gamma+1} \sqsubseteq_\alpha T_P(I_\infty)$. Using again Lemma 4 and the transitivity of $\sqsubseteq_\alpha$ we get that $\forall\gamma < \aleph_1 : I_\gamma \sqsubseteq_\alpha T_P(I_\infty)$. Then, using the first part of Lemma 4, $I_\infty \sqsubseteq_\alpha T_P(I_\infty)$.
Let us prove now $T_P(I_\infty) \sqsubseteq_\alpha I_\infty$. It remains to show

$$I_\infty \parallel \mathcal{F}_\alpha \subseteq T_P(I_\infty) \parallel \mathcal{F}_\alpha \tag{3}$$

as well as

$$T_P(I_\infty)\|\mathcal{T}_\alpha \subseteq I_\infty\|\mathcal{T}_\alpha. \tag{4}$$

Firstly, let us prove that (3) holds and therefore we assume that $I_\infty(A) = \mathcal{F}_\alpha$ for some $A \in H_B$. Then, using the definition of I_∞, we get that for all $i \in \aleph_1$ the following holds:

$$I_i(A) = \mathcal{F}_\alpha \tag{5}$$

Let $A \leftarrow \phi$ be an arbitrary ground instance of P. We prove now that the property $[\![\phi]\!]^{I_\infty} = [\![\phi]\!]^I$ holds. Then, using (5) and the definition of the immediate consequence operator T_P, we get that $\mathcal{F}_\alpha = I_1(A) = \sup\{[\![C]\!]^I;\ A \leftarrow C \in P_G\}$. This implies either $[\![\phi]\!]^I < \mathcal{F}_\alpha$ or $[\![\phi]\!]^I = \mathcal{F}_\alpha$. We consider the first case. Then, using Theorem 3, we get that $[\![\phi]\!]^{I_\infty} = [\![\phi]\!]^I$. In the latter case, using again (5), we get that for all $i \in \aleph_1$ the property $\mathcal{F}_\alpha = I_{i+1}(A) = \sup\{[\![C]\!]^{I_i};\ A \leftarrow C \in P_G\}$ holds. This obviously implies $\forall i \in \aleph_1 : [\![\phi]\!]^{I_i} \leq \mathcal{F}_\alpha$. Then, using Lemma 4 and Theorem 1, we get that $\forall i \in \aleph_1 : [\![\phi]\!]^{I_i} = \mathcal{F}_\alpha$. But then the third part of Theorem 3 finally implies that $[\![\phi]\!]^{I_\infty} = \mathcal{F}_\alpha = [\![\phi]\!]^I$.
Thus the above argumentation implies that for all $A \leftarrow \phi$ in P_G the equation $[\![\phi]\!]^{I_\infty} = [\![\phi]\!]^I$ holds. This implies $\mathcal{F}_\alpha = I_1(A) = \sup\{[\![\phi]\!]^I;\ A \leftarrow \phi \in P_G\} = \sup\{[\![\phi]\!]^{I_\infty};\ A \leftarrow \phi \in P_G\} = T_P(I_\infty)(A)$.
Secondly, let us prove (4) and therefore we assume now that $T_P(I_\infty)(A) = \mathcal{T}_\alpha$ for some $A \in H_B$. Then $\sup\{[\![\phi]\!]^{I_\infty};\ A \leftarrow \phi \in P_G\} = \mathcal{T}_\alpha$. This and Lemma 1 allow us to choose a ground instance $A \leftarrow \phi$ such that $[\![\phi]\!]^{I_\infty} = \mathcal{T}_\alpha$. Then, using Lemma 5, we can choose an ordinal $i_0 \in \aleph_1$ such that $[\![\phi]\!]^{I_{i_0}} = \mathcal{T}_\alpha$. This implies $[\![A]\!]^{I_{i_0+1}} \geq \mathcal{T}_\alpha$. We know $I_{i_0+1} \sqsubseteq_\alpha T_P(I_\infty)$ by Lemma 4 and Lemma 3. But then, using Theorem 1 and the assumption of this case, $[\![A]\!]^{I_{i_0+1}} = \mathcal{T}_\alpha$ must hold. Finally, using the second part of Theorem 3, we get that $I_\infty(A) = \mathcal{T}_\alpha$.
The argumentation above implies that $I_\infty =_\alpha T_P(I_\infty)$. We know that I_∞ does not map to truth values w such that $\mathcal{F}_{\alpha+1} < w \leq \mathcal{T}_{\alpha+1}$. And this obviously implies $I_\infty \sqsubseteq_{\alpha+1} T_P(I_\infty)$. □

Definition 23. Let P be a program. We define by recursion on the countable ordinal α the approximant M_α of P as follows:

$$M_\alpha := \begin{cases} T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\gamma<\alpha} M_\gamma), & \text{if } \begin{array}{l}\forall\gamma < \alpha\forall\zeta < \gamma\,(M_\zeta =_\zeta M_\gamma)\ \&\\ \quad \bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)\end{array} \\ \emptyset, & \text{otherwise} \end{cases}$$

Theorem 4. *Let P be a program, then for all $\alpha \in \aleph_1$ the following holds:*

1. $\forall\gamma < \alpha\,(M_\gamma =_\gamma M_\alpha)$
2. $\bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)$

3. $M_\alpha = T_{P,\alpha}^{\aleph_1}(\bigsqcup_{\gamma<\alpha} M_\gamma)$
4. $M_\alpha \sqsubseteq_{\alpha+1} T_P(M_\alpha)$

Proof. We prove this by induction on α. We assume that the theorem holds for all $\beta < \alpha$ (induction hypothesis). We prove that it holds also for α. Using the induction hypothesis, we get that for every $\beta < \alpha$ the following properties hold $\forall\gamma < \beta : M_\gamma =_\gamma M_\beta$ as well as $M_\beta \sqsubseteq_{\beta+1} T_P(M_\beta)$. Then, using Lemma 6, we get that $\bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)$ (this is 2.). This together with the above definition imply $M_\alpha = T_{P,\alpha}^{\aleph_1}(\bigsqcup_{\gamma<\alpha} M_\gamma)$ (this is 3.). Then, using 2. and 3. and Lemma 7, we get that $M_\alpha \sqsubseteq_{\alpha+1} T_P(M_\alpha)$ (this is 4.). It remains to prove the first statement. We know that for all $\gamma < \alpha$ the property $M_\gamma =_\gamma \bigsqcup_{\gamma'<\alpha} M_{\gamma'} \sqsubseteq_\alpha {}^{\text{(Lemma 4 \& 2.)}} T_{P,\alpha}^{\aleph_1}(\bigsqcup_{\gamma'<\alpha} M_{\gamma'}) =^{3.} M_\alpha$ holds. Then, using that $\sqsubseteq_\alpha$ is stronger than $=_\gamma$, we get that 1. also holds. □

Lemma 8. *Let P be a program. Then there exists an ordinal $\delta \in \aleph_1$ such that*

$$\forall\gamma \geq \delta : M_\gamma \| \mathcal{F}_\gamma = \emptyset \text{ and } M_\gamma \| \mathcal{T}_\gamma = \emptyset. \tag{6}$$

Proof. We define the subset H_B^* of the Herbrand base H_B by $H_B^* := \{A \in H_B;\ \exists\gamma \in \aleph_1 : M_\gamma(A) \in \{\mathcal{F}_\gamma, \mathcal{T}_\gamma\}\}$. Then, using part one of Theorem 4, we know that for every $A \in H_B^*$ there is exactly one γ_A such that $M_{\gamma_A}(A) \in \{\mathcal{F}_{\gamma_A}, \mathcal{T}_{\gamma_A}\}$. Now let us define the function ζ by $\zeta : H_B^* \to \aleph_1 : A \mapsto \gamma_A$. We know that H_B^* is countable. This implies that $\zeta(H_B^*)$ is also countable. Then, using Theorem 2, we know that $\zeta(H_B^*)$ is not cofinal in $\aleph_1$. This obviously implies that there is an ordinal $\delta \in \aleph_1$ such that $\forall A \in H_B^* : \zeta(A) < \delta$. Finally, this ordinal δ satisfies the property (6). □

Definition 24. Let P be a program. The lemma above justifies the definition $\delta_P := \min\{\delta;\ \forall\gamma \geq \delta : M_\gamma \| \mathcal{F}_\gamma = \emptyset \text{ and } M_\gamma \| \mathcal{T}_\gamma = \emptyset\} \in \aleph_1$. This ordinal δ_P is called the *depth* of the program P.

Definition 25. We define the interpretation M_P of a given formula-based logic program P by

$$M_P(A) := \begin{cases} M_{\delta_P}(A), & \text{if } \deg(M_{\delta_P}(A)) < \delta_P \\ 0, & \text{otherwise} \end{cases}.$$

5 Properties of the Interpretation M_P

Proposition 1. *Let P be a program. The interpretation M_P is a fixed point of T_P (i.e., $T_P(M_P) = M_P$).*

Proof. See Theorem 7.1 in [3]. □

Theorem 5. *Let P be a program. The interpretation M_P is a model of P.*

Proof. See Theorem 7.2 in [3]. □

Proposition 2. *Let P be a program, α a countable ordinal and M an arbitrary model of P. Then the following holds:*

$$\forall\beta < \alpha\,(M_\beta =_\beta M) \Rightarrow M_\alpha \sqsubseteq_\alpha M$$

Proof. We assume that $\forall\beta < \alpha\,(M_\beta =_\beta M)$. Definition 22 implies that

$$\bigsqcup_{\beta<\alpha} M_\beta \sqsubseteq_\alpha M\ . \tag{7}$$

Now we prove that the following holds:

$$T_P(M) \sqsubseteq_\alpha M \tag{8}$$

Using Lemma 3 and the assumption above, we get that $\forall\beta < \alpha\,(T_P(M_\beta) =_\beta T_P(M))$. This the assumption above and the fourth part of Theorem 4 imply that $\forall\beta < \alpha : M =_\beta T_P(M)$. But this, together with with the fact that M is a model (i.e., $M(A) \geq T_P(M)(A)$ holds for all atoms $A \in H_U$), implies that (8) holds.

We finish the proof by induction on the ordinal $\gamma \in \Omega$. Using Lemma 3 and (8), we get that $T^{\gamma}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ implies $T^{\gamma+1}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$. Using the first part of Lemma 4, we get for every limit ordinal γ that $\forall \beta < \gamma : T^{\beta}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ implies $T^{\gamma}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$. Then, using (7) and statement 3. of Theorem 4, $M_\alpha = T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ holds. □

Theorem 6. *The interpretation M_P of a given program P is the least of all models of P (i.e., for all models M of P the property $M_P \sqsubseteq_\infty M$ holds).*

Proof. Let M be an arbitrary model of P. Without loss of generality, we assume that $M \neq M_P$. Then let α be the least ordinal such that $\forall \beta < \alpha\, (M_P =_\beta M)$. This implies also $\forall \beta < \alpha\, (M_\beta =_\beta M)$. Then, using Proposition 2, $M_P =_\alpha M_\alpha \sqsubseteq_\alpha M$. The choice of α implies that $M_P \neq_\alpha M$. Then we get that $M_P \sqsubset_\alpha M$ and this finally implies $M_P \sqsubseteq_\infty M$. □

Corollary 1. *Let P be a program. The interpretation M_P is the least of all fixed points of T_P.*

Proof. It is easy to prove that every fixed point of T_P is also a model of P. This together with Proposition 1 and Theorem 6 imply Corollary 1. □

Proposition 3. *There is a countable ordinal $\delta \in \aleph_1$ such that for all programs P of an arbitrary language $\mathcal{L}_{n,m,l,(s_i),(r_i)}$ the property $\delta_P < \delta$ holds. Let $\delta_{\max}$ be the least ordinal such that the above property holds.*

Proof. We know that the set of all signatures $\langle n, m, l, (s_i)_{1\leq i\leq n}, (r_i)_{1\leq i\leq m}\rangle$ is countable. Additionally, we know that the set of all programs of a fixed signature is also countable (Remember that a program is a finite set of rules.). This implies that the set of all programs is countable. Then we get that the image of the function from the set of all programs to $\aleph_1$ given by $P \mapsto \delta_P$ is countable. Then, using Theorem 2, the image of $\delta_{(\cdot)}$ is not cofinal in $\aleph_1$ (i.e., there exists an ordinal $\delta \in \aleph_1$ such that for all programs P the property $\delta_P < \delta$ holds). □

Proposition 4. *The ordinal $\delta_{\max}$ is at least ω^ω.*

Proof. Let $n > 0$ be a natural number. We consider the program P_n consisting of the following rules (where G, H are predicate symbols, f is a function symbol and c is a constant):

$G(x_1, ..., x_{n-1}, f(x_n)) \leftarrow \neg\neg G(x_1, ..., x_{n-1}, x_n)$
For all k provided that $1 \leq k \leq n-1$ the rule:
$G(x_1, ..., x_{k-1}, f(x_k), c, ..., c) \leftarrow \exists x_{k+1}, ..., x_n G(x_1, ..., x_{k-1}, x_k, x_{k+1}, ..., x_n)$
$H \leftarrow \exists x_1, ..., x_n G(x_1, ..., x_n)$

This implies that M_{P_n} maps $G\left(f^{k_1}(c), ..., f^{k_n}(c)\right)$ to $\mathcal{F}_{\sum_{m=1}^{n-1} k_m \omega^{n-m} + k_n \cdot 2}$ and H to $\mathcal{F}_{\omega^n}$. □

At the end of this paper we will prove that the 3-valued interpretation $M_{P,3}$ that results from the infinite-valued model M_P by collapsing all true values to *True* (abbr. $\mathcal{T}$) and all false values to *False* (abbr. $\mathcal{F}$) is also a model in the sense of the following semantics:

Definition 26. *The semantics of formulas with respect to 3-valued interpretations is defined as in Definition 15 except that $[\![\top]\!]^I_h = \mathcal{T}$, $[\![\bot]\!]^I_h = \mathcal{F}$ and*

$$[\![\neg(\phi)]\!]^I_h = \begin{cases} \mathcal{T}, & \text{if } [\![\phi]\!]^I_h = \mathcal{F} \\ \mathcal{F}, & \text{if } [\![\phi]\!]^I_h = \mathcal{T} \\ 0, & \text{otherwise} \end{cases}.$$

The Definition 16 is also suitable in the case of 3-valued interpretations. The truth values are ordered as follows: $\mathcal{F} < 0 < \mathcal{T}$

Proposition 5. *Let P be a program and let* collapse$(\cdot)$ *be the function from W to the set $\{\mathcal{F}, 0, \mathcal{T}\}$ given by $\mathcal{F}_i \mapsto \mathcal{F}$, $0 \mapsto 0$ and $\mathcal{T}_i \mapsto \mathcal{T}$. Moreover, let I be an arbitrary interpretation then* collapse(I) *is the 3-valued interpretation given by* collapse$(I)(A) :=$ collapse$(I(A))$ *for all $A \in H_B$. Then for all formulas ϕ and all assignments h the following holds:*

$$\text{collapse}([\![\phi]\!]_h^I) = [\![\phi]\!]_h^{\text{collapse}(I)}$$

Proof. One can prove this by induction on the structure of ϕ together with Theorem 2. Due to the page constraints, we present only the most interesting case.
Let us assume that $\phi = \forall x(\psi)$ and that the proposition holds for ψ. Obviously, the equation collapse$([\![\phi]\!]_h^I) =$ collapse(inf$\{[\![\psi]\!]_{h[x\mapsto u]}^I;\ u \in H_U\}$) holds. Now we have to consider the following three possible cases, where $I_c :=$ collapse(I) :
Case 1: inf$\{[\![\psi]\!]_{h[x\mapsto u]}^I;\ u \in H_U\} = \mathcal{F}_\alpha$. Then, using Lemma 1, there must be an $u' \in H_U$ such that $[\![\psi]\!]_{h[x\mapsto u']}^I = \mathcal{F}_\alpha$. This implies, using the assumption, that $[\![\psi]\!]_{h[x\mapsto u']}^{I_c} = \mathcal{F}$ and hence $[\![\phi]\!]_h^{I_c} = [\![\forall x(\psi)]\!]_h^{I_c} = \text{inf}\{[\![\psi]\!]_{h[x\mapsto u]}^{I_c};\ u \in H_U\} = \mathcal{F} = \text{collapse}(\mathcal{F}_\alpha) = \text{collapse}([\![\phi]\!]_h^I)$ holds.
Case 2: inf$\{[\![\psi]\!]_{h[x\mapsto u]}^I;\ u \in H_U\} = \mathcal{T}_\alpha$. Then, using the assumption, we get that $[\![\psi]\!]_{h[x\mapsto u]}^{I_c} = \mathcal{T}$ for all $u \in H_U$ and hence $[\![\phi]\!]_h^{I_c} = [\![\forall x(\psi)]\!]_h^{I_c} = \text{inf}\{[\![\psi]\!]_{h[x\mapsto u]}^{I_c};\ u \in H_U\} = \mathcal{T} = \text{collapse}(\mathcal{T}_\alpha) = \text{collapse}([\![\phi]\!]_h^I)$ holds.
Case 3: inf$\{[\![\psi]\!]_{h[x\mapsto u]}^I;\ u \in H_U\} = 0$. We know that H_U is a countable set and hence, using Theorem 2, we get that there must be an $u' \in H_U$ such that $[\![\psi]\!]_{h[x\mapsto u']}^I = 0$. The assumption implies that $[\![\psi]\!]_{h[x\mapsto u']}^{I_c} = 0$ and $0 \leq [\![\psi]\!]_{h[x\mapsto u]}^{I_c}$ for all $u \in H_U$. Hence we get that $[\![\phi]\!]_h^{I_c} = [\![\forall x(\psi)]\!]_h^{I_c} = \text{inf}\{[\![\psi]\!]_{h[x\mapsto u]}^{I_c};\ u \in H_U\} = 0 = \text{collapse}(0) = \text{collapse}([\![\phi]\!]_h^I)$ holds. □

Proposition 6. *Let P be a formula-based logic program. Then the 3-valued interpretation $M_{P,3}$ is a 3-valued model of P.*

Proof. We assume that $A \leftarrow \phi$ is a rule of P. Then, for every assignment h, we get that $[\![\phi]\!]_h^{M_{P,3}} \overset{\text{Proposition 5}}{=} \text{collapse}([\![\phi]\!]_h^{M_P}) \overset{\text{Theorem 6}}{\leq} \text{collapse}([\![A]\!]_h^{M_P}) = [\![A]\!]_h^{M_{P,3}}$ holds. □

Remark 5. The 3-valued model $M_{P,3}$ is not a minimal model in general. Consider the logic program $P = \{P_1 \leftarrow \neg\neg P_1\}$. Then the infinite-valued model M_P maps P_1 to 0 and this implies $M_{P,3}(P_1) = 0$. But the (2-valued) interpretation $\{\langle P_1, \mathcal{F}\rangle\}$ is a model of P and it is less than $M_{P,3}$. The ordering on the 3-valued interpretations is introduced in [2] page 5.

However, Rondogiannis and Wadge prove in [3] that the 3-valued model $M_{P,3}$ of a given normal program P is equal to the 3-valued well-founded model of P and hence, using a result of Przymusinski (Theorem 3.1 of [2]), it is a minimal model of P. In the context of formula-based logic programs we can prove Theorem 7. Before we start with the proof we have to consider the following definition and a lemma that plays an important role in the proof of the theorem.

Definition 27. The *negation degree* $\text{deg}_\neg(\phi)$ of a formula ϕ is defined recursively on the structure of ϕ as follows:

1. If ϕ is an atom, then $\text{deg}_\neg(\phi) := 0$.
2. If $\phi = \psi_1 \circ \psi_2$, then $\text{deg}_\neg(\phi) := \max\{\text{deg}_\neg(\psi_1), \text{deg}_\neg(\psi_2)\}$. $(\circ \in \{\vee, \wedge\})$
3. If $\phi = \boxdot x(\psi)$, then $\text{deg}_\neg(\phi) := \text{deg}_\neg(\psi)$. $(\boxdot \in \{\exists, \forall\})$

Lemma 9. *Let I be an interpretation and $\gamma, \zeta \in \aleph_1$ such that for all $A \in H_B$ the following holds:*

$$I(A) \in [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0]$$

Then for all formulas ϕ such that $\text{deg}_\neg(\phi) \leq 1$ *and all variable assignments h the following holds:*

$$[\![\phi]\!]_h^I \in \begin{cases} [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0], & \textit{if } \text{deg}_\neg(\phi) = 0 \\ [\mathcal{F}_0, \mathcal{F}_{\max\{\gamma,\zeta+1\}}] \cup \{0\} \cup [\mathcal{T}_{\max\{\gamma+1,\zeta\}}, \mathcal{T}_0], & \textit{otherwise} \end{cases}$$

Proof. We prove this by induction on the structure of ϕ.
Case 1: ϕ is an atom. Obviously, if $\phi = \bot$ or $\phi = \top$, then the lemma holds. Otherwise, there is a ground instance $A \in H_B$ of ϕ such that $[\![\phi]\!]^I_h = I(A)$ and the lemma also holds in this case.
Case 2: $\phi = \psi_1 \vee \psi_2$ and the lemma holds for ψ_1 and ψ_2. There is an $i \in \{1, 2\}$ such that $[\![\psi_1]\!]^I_h \leq [\![\psi_i]\!]^I_h$ and $[\![\psi_2]\!]^I_h \leq [\![\psi_i]\!]^I_h$. Then, using $\deg_\neg(\psi_i) \leq \deg_\neg(\phi)$ and $[\![\phi]\!]^I_h = [\![\psi_i]\!]^I_h$, we get that the lemma also holds for ϕ.
Case 3: $\phi = \exists x(\psi)$ and the lemma holds for ψ. Then, using Definition 15, we get that $[\![\phi]\!]^I_h = \sup\{[\![\psi]\!]^I_{h[x\mapsto u]};\ u \in H_U\}$. Let us assume that $\deg_\neg(\phi) = 0$. Then, using the assumption of this case and $\deg_\neg(\psi) \leq \deg_\neg(\phi)$, we get that

$$[\![\psi]\!]^I_{h[x\mapsto u]} \in [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0] \text{ for all } u \in H_U. \tag{9}$$

This implies that the values $\mathcal{F}_\alpha$ and $\mathcal{T}_\beta$ cannot be least upper bounds (for all $\alpha > \gamma$ and for all $\beta > \zeta$). For instance, assume that $\beta > \zeta$ and $\mathcal{T}_\beta$ is a least upper bound. Then, using statement (9), we get that 0 must be an upper bound, and hence this contradicts the assumption that $\mathcal{T}_\beta$ is the least upper bound. This implies $[\![\phi]\!]^I_h = \sup\{[\![\psi]\!]^I_{h[x\mapsto u]};\ u \in H_U\} \in [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0]$ and the lemma holds for ϕ. Now let us assume that $\deg_\neg(\phi) = 1$. This implies that $[\![\psi]\!]^I_{h[x\mapsto u]} \in [\mathcal{F}_0, \mathcal{F}_{\max\{\gamma,\zeta+1\}}] \cup \{0\} \cup [\mathcal{T}_{\max\{\gamma+1,\zeta\}}, \mathcal{T}_0]$ for all $u \in H_U$. Then, using the same argumentation as above, we get that $[\![\phi]\!]^I_h = \sup\{[\![\psi]\!]^I_{h[x\mapsto u]};\ u \in H_U\} \in [\mathcal{F}_0, \mathcal{F}_{\max\{\gamma,\zeta+1\}}] \cup \{0\} \cup [\mathcal{T}_{\max\{\gamma+1,\zeta\}}, \mathcal{T}_0]$ and hence the lemma holds for ϕ.
Case 4: $\phi = \neg(\psi)$ and the lemma holds for ψ. This implies that $\deg_\neg(\psi) = 0$, and hence $[\![\psi]\!]^I_h \in [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0]$. If $[\![\psi]\!]^I_h \in [\mathcal{F}_0, \mathcal{F}_\gamma]$, then $[\![\neg\psi]\!]^I_h \in [\mathcal{T}_{\gamma+1}, \mathcal{T}_0]$. If $[\![\psi]\!]^I_h \in \{0\}$, then $[\![\neg\psi]\!]^I_h \in \{0\}$. If $[\![\psi]\!]^I_h \in [\mathcal{T}_\zeta, \mathcal{T}_0]$, then $[\![\neg\psi]\!]^I_h \in [\mathcal{F}_0, \mathcal{F}_{\zeta+1}]$. Hence, the lemma holds also for ϕ.
We omit the case $\phi = \psi_1 \wedge \psi_2$ (resp. $\phi = \forall x(\psi)$), since it is similar to Case 2 (resp. Case 3). □

Theorem 7. *Let P be a formula-based program such that for every rule $A \leftarrow \phi$ in P the property $\deg_\neg(\phi) \leq 1$ holds. Then the 3-valued model $M_{P,3}$ of the program P is a minimal 3-valued model.*

Proof. Let N_3 be an arbitrary 3-valued model of the program P, such that N_3 is smaller or equal to M_3. This is equivalent to

$$M_{P,3}\|\mathcal{F} \subseteq N_3\|\mathcal{F} \text{ and } N_3\|\mathcal{T} \subseteq M_{P,3}\|\mathcal{T}. \tag{10}$$

Now we have to prove that N_3 is equal to $M_{P,3}$. Note that this holds if and only if both equations $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$ and $N_3\|\mathcal{T} = M_{P,3}\|\mathcal{T}$ hold.
Firstly, we prove that $N_3\|\mathcal{T} = M_{P,3}\|\mathcal{T}$ by contradiction. We assume that

$$M_{P,3}\|\mathcal{T} \setminus N_3\|\mathcal{T} \neq \emptyset. \tag{11}$$

We know that $M_{P,3}\|\mathcal{T} = \bigcup_{\alpha\in\aleph_1} M_P\|\mathcal{T}_\alpha$ and hence, using (11), there must be at least one ordinal $\alpha \in \aleph_1$ such that $M_P\|\mathcal{T}_\alpha \setminus N_3\|\mathcal{T} \neq \emptyset$. This justifies the definition $\alpha_{\min} := \min\{\alpha \in \aleph_1;\ M_P\|\mathcal{T}_\alpha \setminus N_3\|\mathcal{T} \neq \emptyset\}$. Using Theorem 4 we get that $M_{\alpha_{\min}} = T^{\aleph_1}_{P,\alpha_{\min}}(\bigsqcup_{\beta<\alpha_{\min}} M_\beta)$. To improve readability we define $J := \bigsqcup_{\beta<\alpha_{\min}} M_\beta$. It is obviously that $\alpha_{\min} < \delta_P$, and hence Definition 25, Theorem 4, and Definition 20 imply $M_P\|\mathcal{T}_{\alpha_{\min}} = M_{\delta_P}\|\mathcal{T}_{\alpha_{\min}} = M_{\alpha_{\min}}\|\mathcal{T}_{\alpha_{\min}} = \bigcup_{\gamma\in\aleph_1} T^\gamma_{P,\alpha_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}}$. This and the definition of $\alpha_{\min}$ justify the definition $\gamma_{\min} := \min\{\gamma \in \aleph_1;\ T^\gamma_{P,\alpha_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}} \setminus N_3\|\mathcal{T} \neq \emptyset\}$. From Definition 22 and Definition 20 we infer that $0 < \gamma_{\min}$ and $\gamma_{\min}$ is not an infinite limit ordinal, hence $\gamma_{\min}$ is a successor ordinal. We assume that $\gamma_{\min} = \gamma^-_{\min} + 1$. Then, using the definition of $\alpha_{\min}$ and $\gamma_{\min}$, we get that $T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \subseteq N_3\|\mathcal{T}$ for all $\zeta \leq \alpha_{\min}$. Using statement (10) we infer that $T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \subseteq N_3\|\mathcal{F}$ for all $\zeta < \alpha_{\min}$. Hence, the following definition of the infinite-valued interpretation N is well-defined.

$$N(A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \alpha_{\min} \,\&\, A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \\ \mathcal{F}_{\alpha_{\min}}, & \text{if } A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_{\alpha_{\min}} \cap N_3\|\mathcal{F} \\ \mathcal{F}_{\alpha_{\min}+1}, & \text{if } A \in N_3\|\mathcal{F} \setminus \bigcup_{\zeta\leq\alpha_{\min}} T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \\ \mathcal{T}_\zeta, & \text{if } \zeta \leq \alpha_{\min} \,\&\, A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \\ \mathcal{T}_{\alpha_{\min}+1}, & \text{if } A \in N_3\|\mathcal{T} \setminus \bigcup_{\zeta\leq\alpha_{\min}} T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \\ 0, & \text{otherwise} \end{cases} \qquad \text{(for all } A \in H_B\text{)}$$

It is easy to see that

$$T_{P,\alpha_{\min}}^{\gamma_{\min}-1}(J) \sqsubseteq_{\alpha_{\min}} N \text{ and that } N_3 = \text{collapse}(N). \tag{12}$$

Since $T_{P,\alpha_{\min}}^{\gamma_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}} \setminus N_3\|\mathcal{T}$ is not empty, we can pick an A that is contained in this set. Then, together with Definition 18, we get that $\mathcal{T}_{\alpha_{\min}} = T_{P,\alpha_{\min}}^{\gamma_{\min}}(J)(A) = T_P(T_{P,\alpha_{\min}}^{\gamma_{\min}-1}(J))(A) = \sup\{[\![\phi]\!]_I;\ A \leftarrow \phi \in P_G\}$, where $I := T_{P,\alpha_{\min}}^{\gamma_{\min}-1}(J)$. Hence, using Lemma 1, we can pick a rule $A \leftarrow \phi \in P_G$ such that $[\![\phi]\!]_I = \mathcal{T}_{\alpha_{\min}}$. Then, using statement (12), Theorem 1, and Proposition 5, we get that $[\![\phi]\!]_N = \mathcal{T}_{\alpha_{\min}}$ and $[\![\phi]\!]_{N_3} = [\![\phi]\!]_{\text{collapse}(N)} = \text{collapse}([\![\phi]\!]_N) = \mathcal{T}$. Lastly, the fact that N_3 is a model and $A \leftarrow \phi$ is a ground instance of P imply that $N_3(A) = \mathcal{T}$. But this is a contradiction because we have chosen A to be not contained in $N_3\|\mathcal{T}$. Hence, statement (11) must be wrong (i.e., $M_{P,3}\|\mathcal{T} = N_3\|\mathcal{T}$).
Secondly, we show that $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$. Definition 25 implies that $M_{P,3}\|\mathcal{F} = \bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{F}_\zeta$ and $M_{P,3}\|\mathcal{T} = \bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{T}_\zeta$. Then, using (10) and the result of the first part of this proof, we get that $\bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{F}_\zeta \subseteq N_3\|\mathcal{F}$ and $\bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{T}_\zeta = N_3\|\mathcal{T}$. Hence, the following definition of the infinite-valued interpretation N is well-defined and $N_3 = \text{collapse}(N)$.

$$N(A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \delta_P \ \&\ A \in M_{\delta_P}\|\mathcal{F}_\zeta \\ \mathcal{F}_{\delta_P+1}, & \text{if } A \in N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} \\ \mathcal{T}_\zeta, & \text{if } \zeta < \delta_P \ \&\ A \in M_{\delta_P}\|\mathcal{T}_\zeta \\ 0, & \text{otherwise} \end{cases} \qquad \text{(for all } A \in H_B\text{)}$$

Now we are going to prove by transfinite induction on $\zeta \in \aleph_1$ that $T_{P,\delta_P+1}^{\zeta}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$. Obviously, $T_{P,\delta_P+1}^{\zeta}(M_{\delta_P}) =_{\delta_P} N$ for all $\zeta \in \aleph_1$. The Definition of N, Definition 24, and Theorem 4 imply $N\|\mathcal{T}_{\delta_P+1} = \emptyset = M_{\delta_P+1}\|\mathcal{T}_{\delta_P+1} = T_{P,\delta_P+1}^{\aleph_1}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1} = \bigcup_{\gamma<\aleph_1} T_{P,\delta_P+1}^{\gamma}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1}$. Hence, $T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1} \subseteq N\|\mathcal{T}_{\delta_P+1}$ for all $\zeta \in \aleph_1$. It remains to show that $N\|\mathcal{F}_{\delta_P+1} \subseteq T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ for all $\zeta \in \aleph_1$.
Case 1: $\zeta = 0$. It is easy to prove (using Theorem 4, the result of the first part of this proof, and $N_3\|\mathcal{F} \cap N_3\|\mathcal{T} = \emptyset$) that $M_{\delta_P}\|\mathcal{F}_{\delta_P+1} = H_B \setminus (M_{P,3}\|\mathcal{F} \cup M_{P,3}\|\mathcal{T}) \supseteq N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} = N\|\mathcal{F}_{\delta_P+1}$.
Case 2: ζ is a successor ordinal and $T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$. Then, using Definition 20 and Lemma 4, we get that

$$T_P(T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P})) = T_{P,\delta_P+1}^{\zeta}(M_{\delta_P}) \tag{13}$$

and

$$T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1} \subseteq T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}. \tag{14}$$

We will prove that $T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \setminus T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1}$ and $N\|\mathcal{F}_{\delta_P+1}$ are disjoint. This, using $T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$ and statement (14), implies that $N\|\mathcal{F}_{\delta_P+1} \subseteq T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ and we have proved this case. Therefore, we choose an arbitrary $A \in T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \setminus T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1}$. Hence, using Lemma 4, we get that $\mathcal{F}_{\delta_P+1} < T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})(A)$. This, together with (13) and Definition 18, implies that there must be a rule $A \leftarrow \phi \in P_G$ such that $\mathcal{F}_{\delta_P+1} < [\![\phi]\!]_I$, where I is given by $I := T_{P,\delta_P+1}^{\zeta-1}(M_{\delta_P})$. Then, using the assumption $I \sqsubseteq_{\delta_P+1} N$ and Theorem 1, we get that $\mathcal{F}_{\delta_P+1} < [\![\phi]\!]_N$. We know that for all atoms $C \in H_B$ the image $N(C)$ is an element of $[F_0, F_{\delta_P+1}] \cup \{0\} \cup [T_{\delta_P}, T_0]$. Then Lemma 9 and the fact that $\deg_\neg(\phi) \leq 1$ imply $0 \leq [\![\phi]\!]_N$. Hence, using Proposition 5, $N_3 = \text{collapse}(N)$ and N_3 is a model of P, we get that $0 \leq [\![\phi]\!]_{N_3} \leq N_3(A)$. Finally, this implies $A \notin N_3\|\mathcal{F} \supseteq N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} = N\|\mathcal{F}_{\delta_p+1}$.
Case 3: $\zeta > 0$ is a limit ordinal and $T_{P,\delta_P+1}^{\gamma}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$ for all $\gamma < \zeta$. This implies $N\|\mathcal{F}_{\delta_P+1} \subseteq T_{P,\delta_P+1}^{\gamma}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ for all $\gamma < \zeta$. Hence, using Definition 20, we get that $T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} = \bigcap_{\gamma\in\zeta} T_{P,\delta_P+1}^{\gamma}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \supseteq N\|\mathcal{F}_{\delta_P+1}$.
The above transfinite induction shows that $N\|\mathcal{F}_{\delta_P+1} \subseteq \bigcap_{\zeta\in\aleph_1} T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$. Then, using that $M_{\delta_P+1}\|\mathcal{F}_{\delta_P+1} = \emptyset$ and $M_{\delta_P+1}\|\mathcal{F}_{\delta_P+1} = \bigcap_{\zeta\in\aleph_1} T_{P,\delta_P+1}^{\zeta}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$, we get that $\emptyset = N\|\mathcal{F}_{\delta_P+1} = N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F}$ (see definition of N above). Last of all, using the assumption (10), we get that $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$. □

6 Summary and Future Work

We have shown that every formula-based logic program P has a least infinite-valued model M_P with respect to the ordering $\sqsubseteq_\infty$ given on the set of all infinite-valued interpretations. We have presented how to construct the model M_P with the help of the immediate consequence operator T_P and have shown that M_P is also the least of all fixed points of the operator T_P. Moreover, we have considered the 3-valued interpretation $M_{P,3}$ and have proven that it is a 3-valued model of the program P. Furthermore, we have observed a restricted class of formula-based programs such that the associated 3-valued models are even minimal models.
There are some aspects of this paper that we feel should be further investigated. Firstly, we believe that the main results of this work also hold in Zermelo-Fraenkel axiomatic set theory without the Axiom of Choice (ZF). For instance, we could use the class of all ordinals Ω instead of the cardinal $\aleph_1$ in Theorem 3. Secondly, we have proven that the ordinal $\delta_{\max}$ is at least ω^ω, but on the other hand we do not know a program P such that $\omega^\omega < \delta_P$. So, one could assume that $\delta_{\max} = \omega^\omega$. Thirdly, the negation-as-failure rule is sound for M_P (respectively, $M_{P,3}$) when we are dealing with a normal program P. Within the context of formula-based programs we think it would be fruitful to investigate the rule of definitional reflection presented in [4] instead of negation-as-failure. Lastly, we believe that the presented theory can be useful in the areas of databases and data mining. We are looking forward to collaborate with research groups specializing in these areas.

Acknowledgements. This work has been financed through a grant made available by the Carl Zeiss Foundation. The author is grateful to Prof. Dr. Peter Schroeder-Heister, Hans-Joerg Ant, M. Comp. Sc., and three anonymous reviewers for helpful comments and suggestions. The final preparation of the manuscript was supported by DFG grant Schr275/16-1.

References

1. Jech, T., *Set Theory. The Third Millennium Edition, Revised and Expanded*, Springer-Verlag, Berlin, Heidelberg, New York, 2002, Page 49
2. Przymusinski, T., *Every Logic Program Has a Natural Stratification And an Iterated Least Fixed Point Model,* Eighth ACM Symposium on Principles of Database Systems, Page 11-21, 1989
3. Rondogiannis, P. and Wadge, W., *Minimum Model Semantics for Logic Programs with Negation-as-Failure*, ACM Transactions on Computational Logic, Vol. 6, No. 2, April 2005, Page 441-467
4. Schroeder-Heister, P., *Rules of definitional reflection*, Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (Montreal 1993), Los Alamitos 1993, Page 222-232
5. Van Gelder, A., Ross, K. and Schlipf, J., *The Well-Founded Semantics for General Logic Programs*, Journal of the ACM, Vol. 38, No. 3, July 1991, pp. 620-650

Lifted Unit Propagation for Effective Grounding

Pashootan Vaezipoor[1], David Mitchell[1], and Maarten Mariën*[2]

[1] Department of Computing Science, Simon Fraser University, Canada {pva6,mitchell}@cs.sfu.ca
[2] Department of Computer Science, Katholieke Universiteit Leuven, Belgium maartenm@cs.kuleuven.be

Abstract. A grounding of a formula ϕ over a given finite domain is a ground formula which is equivalent to ϕ on that domain. Very effective propositional solvers have made grounding-based methods for problem solving increasingly important, however for realistic problem domains and instances, the size of groundings is often problematic. A key technique in ground (e.g., SAT) solvers is unit propagation, which often significantly reduces ground formula size even before search begins. We define a "lifted" version of unit propagation which may be carried out prior to grounding, and describe integration of the resulting technique into grounding algorithms. We describe an implementation of the method in a bottom-up grounder, and an experimental study of its performance.

1 Introduction

Grounding is central in many systems for solving combinatorial problems based on declarative specifications. In grounding-based systems, a "grounder" combines a problem specification with a problem instance to produce a ground formula which represents the solutions for the instance. A solution (if there is one) is obtained by sending this formula to a "ground solver", such as a SAT solver or propositional answer set programming (ASP) solver. Many systems have specifications given in extensions or restrictions of classical first order logic (FO), including: IDP [WMD08c], MXG [Moh04], Enfragmo [ATÜ+10,AWTM11], ASPPS [ET06], and Kodkod [TJ07]. Specifications for ASP systems, such as DLV [LPF+06] and clingo [GKK+08], are (extended) normal logic programs under stable model semantics.

Here our focus is grounding specifications in the form of FO formulas. In this setting, formula ϕ constitutes a specification of a problem (e.g., graph 3-colouring), and a problem instance is a finite structure $\mathcal{A}$ (e.g., a graph). The grounder, roughly, must produce a ground formula ψ which is logically equivalent to ϕ over the domain of $\mathcal{A}$. Then ψ can be transformed into a propositional CNF formula, and given as input to a SAT solver. If a satisfying assignment is found, a solution to $\mathcal{A}$ can be constructed from it. ASP systems use an analogous process.

A "naive" grounding of ϕ over a finite domain A can be obtained by replacing each sub-formula of the form $\exists x\, \psi(x)$ with $\bigvee_{a \in A} \psi(\tilde{a})$, where $\tilde{a}$ is a constant symbol which denotes domain element a, and similarly replacing each subformula $\forall x\, \psi(x)$ with a conjunction. For a fixed FO formula ϕ, this can be done in time polynomial in $|A|$. Most grounders use refinements of this method, implemented top-down or bottom-up, and perform well on simple benchmark problems and small instances. However, as we tackle more realistic problems with complex specifications and instances having large domains, the groundings produced can become prohibitively large. This can be the case even when the formulas are "not too hard". That is, the system performance is poor because of time spent generating and manipulating this large ground formula, yet an essentially equivalent but smaller formula can be solved in reasonable time. This work represents one direction in our group's efforts to develop techniques which scale effectively to complex specifications and large instances.

Most SAT solvers begin by executing unit propagation (UP) on the input formula (perhaps with other "pre-processing"). This initial application of UP often eliminates a large number of variables and clauses, and is done very fast. However, it may be too late: the system has already spent a good deal of time generating large but rather uninteresting (parts of) ground formulas, transforming them to CNF, moving them from the grounder to the SAT solver, building the SAT solver's data structures, etc. This suggests trying to execute a process similar to UP *before or during grounding*.

One version of this idea was introduced in [WMD08b,WMD10]. The method presented there involves computing a *symbolic* and *incomplete* representation of the information that UP could derive, obtained

* This author's contributions to this paper were made while he was a post-doctoral fellow at SFU.

from ϕ alone without reference to a particular instance structure. For brevity, we refer to that method as GWB, for "Grounding with Bounds". In [WMD08b,WMD10], the top-down grounder GIDL [WMD08a] is modified to use this information, and experiments indicate it significantly reduces the size of groundings without taking unreasonable time.

An alternate approach is to construct a *concrete* and *complete* representation of the information that UP can derive about a grounding of ϕ over $\mathcal{A}$, and use this information during grounding to reduce grounding size. This paper presents such a method, which we call *lifted unit propagation* (LUP). (The authors of the GWB papers considered this approach also [DW08], but to our knowledge did not implement it or report on it. The relationship between GWB and LUP is discussed further in Section 7.) The LUP method is roughly as follows.

1. Modify instance structure $\mathcal{A}$ to produce a new (partial) structure which contains information equivalent to that derived by executing UP on the CNF formula obtained from a grounding of ϕ over $\mathcal{A}$. We call this new partial structure the LUP structure for ϕ and $\mathcal{A}$, denoted $\mathcal{LUP}(\phi,\mathcal{A})$.
2. Run a modified (top-down or bottom-up) grounding algorithm which takes as input, ϕ and $\mathcal{LUP}(\phi,\mathcal{A})$, and produces a grounding of ϕ over $\mathcal{A}$.

The modification in step 2 relies on the idea that a tuple in $\mathcal{LUP}(\phi,\mathcal{A})$ indicates that a particular subformula has the same (known) truth value in every model. Thus, that subformula may be replaced with its truth value. The CNF formula obtained by grounding over $\mathcal{LUP}(\phi,\mathcal{A})$ is at most as large as the formula that results from producing the naive grounding and then executing UP on it. Sometimes it is much smaller than this, because the grounding method naturally eliminates some autark sub-formulas which UP does not eliminate, as explained in Sections 3 and 6.

We compute the LUP structure by constructing, from ϕ, an inductive definition of the relations of the LUP structure for ϕ and $\mathcal{A}$ (see Section 4). We implemented a semi-naive method for evaluating this inductive definition, based on relational algebra, within our grounder Enfragmo. (We also computed these definitions using the ASP grounders gringo and DLV, but these were not faster.)

For top-down grounding (see Section 3), we modify the naive recursive algorithm to check the derived information in $\mathcal{LUP}(\phi, \mathcal{A})$ at the time of instantiating each sub-formula of ϕ. This algorithm is presented primarily for expository purposes, and is similar to the modified top-down algorithm used for GWB in GIDL.

For bottom-up grounding (see Section 5), we revise the bottom-up grounding method based on extended relational algebra described in [MTHM06,PLTG07], which is the basis of grounders our group has been developing. The change required to ground using $\mathcal{LUP}(\phi, \mathcal{A})$ is a simple revision to the base case.

In Section 6 we present an experimental evaluation of the performance of our grounder Enfragmo with LUP. This evaluation is limited by the fact that our LUP implementation does not support specifications with arithmetic or aggregates, and a shortage of interesting benchmarks which have natural specifications without these features. Within the limited domains we have tested to date, we found:

1. CNF formulas produced by Enfragmo with LUP are always smaller than the result of running UP on the CNF formula produced by Enfragmo without LUP, and in some cases much smaller.
2. CNF formulas produced by Enfragmo with LUP are always smaller than the ground formulas produced by GIDL, with or without GWB turned on.
3. Grounding over $\mathcal{LUP}(\phi, \mathcal{A})$ is always slower than grounding without, but CNF transformation with LUP is almost always faster than without.
4. Total solving time for Enfragmo with LUP is sometimes significantly less than that of Enfragmo without LUP, but in other cases is somewhat greater.
5. Enfragmo with LUP and the SAT solver MiniSat always runs faster than the IDP system (GIDL with ground solver MINISAT(ID)), with or without the GWB method turned on in GIDL.

Determining the extent to which these observations generalize is future work.

2 FO Model Expansion and Grounding

A natural formalization of combinatorial search problems and their specifications is as the logical task of model expansion (MX) [MT11]. Here, we define MX for the special case of FO. Recall that a structure $\mathcal{B}$

for vocabulary $\sigma \cup \varepsilon$ is an expansion of σ-structure $\mathcal{A}$ iff $\mathcal{A}$ and $\mathcal{B}$ have the same domain ($A = B$), and interpret their common vocabulary identically, i.e., for each symbol R of σ, $R^{\mathcal{B}} = R^{\mathcal{A}}$. Also, if $\mathcal{B}$ is an expansion of σ-structure $\mathcal{A}$, then $\mathcal{A}$ is the reduct of $\mathcal{B}$ defined by σ.

Definition 1 (Model Expansion for FO).

Given: A FO formula ϕ on vocabulary $\sigma \cup \varepsilon$ and a σ-structure $\mathcal{A}$,
Find: an expansion $\mathcal{B}$ of $\mathcal{A}$ that satisfies ϕ.

In the present context, the formula ϕ constitutes a problem specification, the structure $\mathcal{A}$ a problem instance, and expansions of $\mathcal{A}$ which satisfy ϕ are solutions for $\mathcal{A}$. Thus, we call the vocabulary of $\mathcal{A}$, the *instance* vocabulary, denoted by σ, and ε the *expansion* vocabulary. We sometimes say ϕ is $\mathcal{A}$-satisfiable if there exists an expansion $\mathcal{B}$ of $\mathcal{A}$ that satisfies ϕ.

Example 1. Consider the following formula ϕ:

$$\forall x[(R(x) \vee B(x) \vee G(x)) \wedge \neg(R(x) \wedge B(x)) \wedge \neg(R(x) \wedge G(x)) \wedge \neg(B(x) \wedge G(x))]$$
$$\wedge\ \forall x \forall y[E(x,y) \supset (\neg(R(x) \wedge R(y)) \wedge \neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))].$$

A finite structure $\mathcal{A}$ over vocabulary $\sigma = \{E\}$, where E is a binary relation symbol, is a graph. Given graph $\mathcal{A} = \mathcal{G} = (V; E)$, there is an expansion $\mathcal{B}$ of $\mathcal{A}$ that satisfies ϕ, iff $\mathcal{G}$ is 3-colourable. So ϕ constitutes a specification of the problem of graph 3-colouring. To illustrate:

$$\underbrace{\overbrace{(V; E^{\mathcal{A}}}^{\mathcal{A}}, R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi$$

An interpretation for the expansion vocabulary $\varepsilon := \{R, B, G\}$ given by structure $\mathcal{B}$ is a colouring of $\mathcal{G}$, and the proper 3-colourings of $\mathcal{G}$ are the interpretations of ε in structures $\mathcal{B}$ that satisfy ϕ.

2.1 Grounding for Model Expansion

Given ϕ and $\mathcal{A}$, we want to produce a CNF formula (for input to a SAT solver), which represents the solutions to $\mathcal{A}$. We do this in two steps: grounding, followed by transformation to CNF. The grounding step produces a ground formula ψ which is equivalent to ϕ over expansions of $\mathcal{A}$. To produce ψ, we bring domain elements into the syntax by expanding the vocabulary with a new constant symbol for each domain element. For A, the domain of $\mathcal{A}$, we denote this set of constants by $\tilde{A}$. For each $a \in A$, we write $\tilde{a}$ for the corresponding symbol in $\tilde{A}$. We also write $\tilde{\bar{a}}$, where $\bar{a}$ is a tuple.

Definition 2 (Grounding of ϕ over $\mathcal{A}$). *Let ϕ be a formula of vocabulary $\sigma \cup \varepsilon$, $\mathcal{A}$ be a finite σ-structure, and ψ be a ground formula of vocabulary μ, where $\mu \supseteq \sigma \cup \varepsilon \cup \tilde{A}$. Then ψ is a grounding of ϕ over $\mathcal{A}$ if and only if:*

1. *if ϕ is $\mathcal{A}$-satisfiable then ψ is $\mathcal{A}$-satisfiable;*
2. *if $\mathcal{B}$ is a μ-structure which is an expansion of $\mathcal{A}$ and gives $\tilde{A}$ the intended interpretation, and $\mathcal{B} \models \psi$, then $\mathcal{B} \models \phi$.*

We call ψ a reduced grounding if it contains no symbols of the instance vocabulary σ.

Definition 2 is a slight generalization of that used in [MTHM06,PLTG07], in that it allows ψ to have vocabulary symbols not in $\sigma \cup \varepsilon \cup \tilde{A}$. This generalization allows us to apply a Tseitin-style CNF transformation in such a way that the resulting CNF formula is still a grounding of ϕ over $\mathcal{A}$. If $\mathcal{B}$ is an expansion of $\mathcal{A}$ satisfying ψ, then the reduct of $\mathcal{B}$ defined by $\sigma \cup \varepsilon$ is an expansion of $\mathcal{A}$ that satisfies ϕ. For the remainder of the paper, we assume that ϕ is in negation normal form (NNF), i.e., negations are applied only to atoms. Any formula may be transformed in linear time to an equivalent formula in NNF.

Algorithm 1 produces the "naive grounding" of ϕ over $\mathcal{A}$ mentioned in the introduction. A substitution is a set of pairs (x/a), where x is a variable and a a constant symbol. If θ is a substitution, then $\phi[\theta]$ denotes

Algorithm 1 Top-Down Naive Grounding of NNF formula ϕ over $\mathcal{A}$

$$\mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\theta)=\begin{cases} P(\bar{x})[\theta] & \text{if } \phi \text{ is an atom } P(\bar{x}) \\ \neg P(\bar{x})[\theta] & \text{if } \phi \text{ is a negated atom } \neg P(\bar{x}) \\ \bigwedge_i \mathrm{NaiveGnd}_{\mathcal{A}}(\psi_i,\theta) & \text{if } \phi = \bigwedge_i \psi_i \\ \bigvee_i \mathrm{NaiveGnd}_{\mathcal{A}}(\psi_i,\theta) & \text{if } \phi = \bigvee_i \psi_i \\ \bigwedge_{a\in A} \mathrm{NaiveGnd}_{\mathcal{A}}(\psi,[\theta\cup(x/\tilde{a})]) & \text{if } \phi = \forall x\ \psi \\ \bigvee_{a\in A} \mathrm{NaiveGnd}_{\mathcal{A}}(\psi,[\theta\cup(x/\tilde{a})]) & \text{if } \phi = \exists x\ \psi \end{cases}$$

the result of substituting constant symbol a for each free occurrence of variable x in ϕ, for every (x/a) in θ. We allow conjunction and disjunction to be connectives of arbitrary arity. That is $(\wedge\ \phi_1\ \phi_2\ \phi_3)$ is a formula, not just an abbreviation for some parenthesization of $(\phi_1 \wedge \phi_2 \wedge \phi_3)$. The initial call to Algorithm 1 is $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\emptyset)$, where $\emptyset$ is the empty substitution.

The ground formula produced by Algorithm 1 is not a grounding of ϕ over $\mathcal{A}$ (according to Definition 2), because it does not take into account the interpretations of σ given by $\mathcal{A}$. To produce a grounding of ϕ over $\mathcal{A}$, we may conjoin a set of atoms giving that information. In the remainder of the paper, we write $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi)$ for the result of calling $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\emptyset)$ and conjoining ground atoms to it to produce a grounding of ϕ over $\mathcal{A}$. We may also produce a reduced grounding from $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\emptyset)$ by "evaluating out" all atoms of the instance vocabulary. The groundings produced by algorithms described later in this paper can be obtained by simplifying out certain sub-formulas of $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi)$.

2.2 Transformation to CNF and Unit Propagation

To transform a ground formula to CNF, we employ the method of Tseitin [Tse68] with two modifications. The method, usually presented for propositional formulas, involves adding a new atom corresponding to each sub-formula. Here, we use a version for ground FO formulas, so the resulting CNF formula is also a ground FO formula, over vocabulary $\tau = \sigma \cup \varepsilon \cup \tilde{A} \cup \omega$, where ω is a set of new relation symbols which we call "Tseitin symbols". To be precise, ω consists of a new k-ary relation symbol $\lceil\psi\rceil$ for each subformula ψ of ϕ with k free variables. We also formulate the transformation for formulas in which conjunction and disjunction may have arbitrary arity.

Let $\gamma = \mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\emptyset)$. Each subformula α of γ is a grounding over $\mathcal{A}$ of a substitution instance $\psi(\bar{x})[\theta]$, of some subformula ψ of ϕ with free variables $\bar{x}$. To describe the CNF transformation, it is useful to think of labelling the subformulas of γ during grounding as follows. If α is a grounding of formula $\psi(\bar{x})[\theta]$, label α with the ground atom $\lceil\psi\rceil(\bar{x})[\theta]$. To minimize notation, we will denote this atom by $\widehat{\alpha}$, setting $\widehat{\alpha}$ to α if α is an atom. Now, we have for each sub-formula α of the ground formula ψ, a unique ground atom $\widehat{\alpha}$, and we carry out the Tseitin transformation to CNF using these atoms.

Definition 3. *For ground formula ψ, we denote by $\mathrm{CNF}(\psi)$ the following set of ground clauses. For each sub-formula α of ψ of form $(\wedge_i\ \alpha_i)$, include in $\mathrm{CNF}(\psi)$ the set of clauses $\{(\neg\widehat{\alpha} \vee \widehat{\alpha_i})\} \cup \{(\vee_i \neg\widehat{\alpha_i} \vee \widehat{\alpha})\}$, and similarly for the other connectives.*

If ψ is a grounding of ϕ over $\mathcal{A}$, then CNF(ψ) is also. The models of ψ are exactly the reducts of the models of CNF(ψ) defined by $\sigma \cup \varepsilon \cup \tilde{A}$. CNF($\psi$) can trivially be viewed as a propositional CNF formula. This propositional formula can be sent to a SAT solver, and if a satisfying assignment is found, a model of ϕ which is an expansion of $\mathcal{A}$ can be constructed from it.

Definition 4 (UP(γ)). *Let γ be a ground FO formula in CNF. Define $\mathrm{UP}(\gamma)$, the result of applying unit propagation to γ, to be the fixed point of the following operation:*

> *If γ contains a unit clause (l), delete from each clause of γ every occurrence of $\neg l$, and delete from γ every clause containing l.*

Now, $\mathrm{CNF}(\mathrm{NaiveGND}_{\mathcal{A}}(\phi))$ is the result of producing the naive grounding of ϕ over $\mathcal{A}$, and transforming it to CNF in the standard way, and $\mathrm{UP}(\mathrm{CNF}(\mathrm{NaiveGND}_{\mathcal{A}}(\phi)))$ is the formula obtained after simplifying it by executing unit propagation. These two formulas provide reference points for measuring the reduction in ground formula size obtained by LUP.

3 Bound Structures and Top-down Grounding

We present grounding algorithms, in this section and in Section 4, which produce groundings of ϕ over a class of partial structures, which we call bound structures, related to $\mathcal{A}$. The structure $\mathcal{LUP}(\phi,\mathcal{A})$ is a particular bound structure. In this section, we define partial structures and bound structures, and then present a top-down grounding algorithm. The formalization of bound structures here, and of $\mathcal{LUP}(\phi,\mathcal{A})$ in Section 4, are ours, although a similar formalization was implicit in [DW08].

3.1 Partial Structures and Bound Structures

A relational τ-structure $\mathcal{A}$ consists of a domain A together with a relation $R^{\mathcal{A}} \subset A^k$ for each k-ary relation symbol of τ. To talk about partial structures, in which the interpretation of a relation symbol may be only partially defined, it is convenient to view a structure in terms of the characteristic functions of the relations. Partial τ-structure $\mathcal{A}$ consists of a domain A together with a k-ary function $\chi_R^{\mathcal{A}} : A^k \to \{\top, \bot, \infty\}$, for each k-ary relation symbol R of τ. Here, as elsewhere, $\top$ denotes true, $\bot$ denotes false, and ∞ denotes undefined. If each of these characteristic functions is total, then $\mathcal{A}$ is total. We may sometimes abuse terminology and call a relation partial, meaning the characteristic function interpreting the relation symbol in question is partial.

Assume the natural adaptation of standard FO semantics the to the case of partial relations, e.g. with Kleene's 3-valued semantics [Kle52]. For any (total) τ-structure $\mathcal{B}$, each τ-sentence ϕ is either true or false in $\mathcal{B}$ ($\mathcal{B} \models \phi$ or $\mathcal{B} \not\models \phi$), and each τ-formula $\phi(\bar{x})$ with free variables $\bar{x}$, defines a relation

$$\phi^{\mathcal{B}} = \{\bar{a} \in A^{|\bar{x}|} : \mathcal{B} \models \phi(\bar{x})[\bar{x}/\bar{a}]\}. \tag{1}$$

Similarly, for any partial τ-structure, each τ-sentence is either true, false or undetermined in $\mathcal{B}$, and each τ-formula $\phi(\bar{x})$ with free variables $\bar{x}$ defines a partial function

$$\chi_\phi^{\mathcal{A}} : A^k \to \{\top, \bot, \infty\}. \tag{2}$$

In the case $\chi_\phi^{\mathcal{A}}$ is total, it is the characteristic function of the relation (1).

There is a natural partial order on partial structures for any vocabulary τ, which we may denote by $\leq$, where $\mathcal{A} \leq \mathcal{B}$ iff $\mathcal{A}$ and $\mathcal{B}$ agree at all points where they are both defined, and $\mathcal{B}$ is defined at every point $\mathcal{A}$ is. If $\mathcal{A} \leq \mathcal{B}$, we may say that $\mathcal{B}$ is a strengthening of $\mathcal{A}$. When convenient, if the vocabulary of $\mathcal{A}$ is a proper subset of that of $\mathcal{B}$, we may still call $\mathcal{B}$ a strengthening of $\mathcal{A}$, taking $\mathcal{A}$ to leave all symbols not in its vocabulary, completely undefined. We will call $\mathcal{B}$ a conservative strengthening of $\mathcal{A}$ with respect to formula ϕ if $\mathcal{B}$ is a strengthening of $\mathcal{A}$ and in addition every total structure which is a strengthening of $\mathcal{A}$ and a model of ϕ is also a strengthening of $\mathcal{B}$. (Intuitively, we could ground ϕ over $\mathcal{B}$ instead of $\mathcal{A}$, and not lose any intended models.)

The specific structures of interest are over a vocabulary expanding the vocabulary of ϕ in a certain way. We will call a vocabulary τ a Tseitin vocabulary for ϕ if it contains, in addition to the symbols of ϕ, the set ω of Tseitin symbols for ϕ. We call a τ-structure a "Tseitin structure for ϕ" if the interpretations of the Tseitin symbols respect the special role of those symbols in the Tseitin transformation. For example, if α is $\alpha_1 \wedge \alpha_2$, then $\widehat{\alpha}^{\mathcal{A}}$ must be true iff $\widehat{\alpha_1}^{\mathcal{A}} = \widehat{\alpha_2}^{\mathcal{A}} = true$. The vocabulary of the formula $\mathrm{CNF}(\mathrm{NaiveGnd}_{\mathcal{A}}(\phi))$ is a Tseitin vocabulary for ϕ, and every model of that formula is a Tseitin structure for ϕ.

Definition 5 (Bound Structures). *Let ϕ be a formula, and $\mathcal{A}$ be a structure for a sub-set of the vocabulary of ϕ. A bound structure for ϕ and $\mathcal{A}$ is a partial Tseitin structure for ϕ that is a conservative strengthening of $\mathcal{A}$ with respect to ϕ.*

Intuitively, a bound structure provides a way to represent the information from the instance together with additional information, including information about the Tseitin symbols in a grounding of ϕ, that we may derive (by any means), provided that information does not eliminate any intended models.

Let τ be the minimum vocabulary for bound structures for ϕ and $\mathcal{A}$. The bound structures for ϕ and $\mathcal{A}$ with vocabulary τ form a lattice under the partial order $\leq$, with $\mathcal{A}$ the minimum element. The maximum element is defined exactly for the atoms of $\mathrm{CNF}(\mathrm{NaiveGnd}_{\mathcal{A}}(\phi))$ which have the same truth value in every Tseitin τ-structure that satisfies ϕ. This is the structure produced by "Most Optimum Propagator" in [WMD10]).

Definition 6 (Grounding over a bound structure). *Let $\hat{\mathcal{A}}$ be a bound structure for ϕ and $\mathcal{A}$. A formula ψ, over a Tseitin vocabulary for ϕ which includes $\tilde{A}$, is a grounding of ϕ over $\hat{\mathcal{A}}$ iff*

1. *if there is a total strengthening of $\hat{\mathcal{A}}$ that satisfies ϕ, then there is a one that satisfies ψ;*
2. *if $\mathcal{B}$ is a total Tseitin structure for ϕ which strengthens $\hat{\mathcal{A}}$, gives $\tilde{A}$ the intended interpretation and satisfies ψ, then it satisfies ϕ.*

A grounding ψ of ϕ over $\hat{\mathcal{A}}$ need not be a grounding of ϕ over $\mathcal{A}$. If we conjoin with ψ ground atoms representing the information contained in $\hat{\mathcal{A}}$, then we do obtain a grounding of ϕ over $\mathcal{A}$. In practice, we send just CNF(ψ) to the SAT solver, and if a satisfying assignment is found, add the missing information back in at the time we construct a model for ϕ.

3.2 Top-down Grounding over a Bound Structure

Algorithm 2 produces a grounding of ϕ over a bound structure $\hat{\mathcal{A}}$ for $\mathcal{A}$. Gnd and $Simpl$ are defined by mutual recursion. Gnd performs expansions and substitutions, while $Simpl$ performs lookups in $\hat{\mathcal{A}}$ to see if the grounding of a sub-formula may be left out. $Eval$ provides the base cases, evaluating ground atoms over $\sigma \cup \varepsilon \cup \tilde{A} \cup \omega$ in $\hat{\mathcal{A}}$.

Algorithm 2 Top-Down Grounding over Bound Structure $\hat{\mathcal{A}}$ for ϕ and $\mathcal{A}$

$$Gnd_{\hat{\mathcal{A}}}(\phi,\theta) = \begin{cases} Eval_{\hat{\mathcal{A}}}(P,\theta) & \phi \text{ is an atom } P(\bar{x}) \\ \neg Eval_{\hat{\mathcal{A}}}(P,\theta) & \phi \text{ is a negated atom } \neg P(\bar{x}) \\ \bigwedge_i Simpl_{\hat{\mathcal{A}}}(\psi_i,\theta) & \phi = \bigwedge_i \psi_i \\ \bigvee_i Simpl_{\hat{\mathcal{A}}}(\psi_i,\theta) & \phi = \bigvee_i \psi_i \\ \bigwedge_{a\in A} Simpl_{\hat{\mathcal{A}}}(\psi,\theta \cup (x/\tilde{a})) & \phi = \forall x\ \psi \\ \bigvee_{a\in A} Simpl_{\hat{\mathcal{A}}}(\psi,\theta \cup (x/\tilde{a})) & \phi = \exists x\ \psi \end{cases}$$

$$Eval_{\hat{\mathcal{A}}}(P,\theta) = \begin{cases} \top & \hat{\mathcal{A}} \models P[\theta] \\ \bot & \hat{\mathcal{A}} \models \neg P[\theta] \\ P(\bar{x})[\theta] & \text{o.w} \end{cases}$$

$$Simpl_{\hat{\mathcal{A}}}(\psi,\theta) = \begin{cases} \top & \hat{\mathcal{A}} \models \lceil\psi\rceil[\theta] \\ \bot & \hat{\mathcal{A}} \models \neg\lceil\psi\rceil[\theta] \\ Gnd_{\hat{\mathcal{A}}}(\psi,\theta) & \text{o.w} \end{cases}$$

The stronger $\hat{\mathcal{A}}$ is, the smaller the ground formula produced by Algorithm 2. If we set $\hat{\mathcal{A}}$ to be undefined everywhere (i.e., to just give the domain), then Algorithm 2 produces $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi,\emptyset)$. If $\hat{\mathcal{A}}$ is set to $\mathcal{A}$, we get the reduced grounding obtained by evaluating instance symbols out of $\mathrm{NaiveGnd}_{\mathcal{A}}(\phi)$.

Proposition 1. *Algorithm 2 produces a grounding of ϕ over $\hat{\mathcal{A}}$.*

3.3 Autarkies and Autark Subformulas

In the literature, an *autarky* [MS85] is informally a "self-sufficient" model for some clauses which does not affect the remaining clauses of the formula. An autark subformula is a subformula which is satisfied by an autarky. To see how an autark subformula may be produced during grounding, let $\lambda = \gamma_1 \vee \gamma_2$ and imagine that the value of subformula γ_1 is true according to our bound structure. Then λ will be true, regardless of the value of γ_2, and the grounder will replace its subformula with its truth value, whereas in the case of naive grounding, the grounder does not have that information during the grounding. So it generates the set of clauses for this subformula as: $\{(\neg\lambda \vee \gamma_1 \vee \gamma_2), (\neg\gamma_1 \vee \lambda), (\neg\gamma_2 \vee \lambda)\}$. Now the propagation of the truth value of λ_1 and subsequently λ, results in elimination of all the three clauses, but the set of clauses

generated for γ_2 will remain in the CNF formula. We call γ_2 and the clauses made from that subformula autarkies.

The example suggests that this is a common phenomena and that the number of autarkies might be quite large in many groundings, as will be seen in Section 6.

4 Lifted Unit Propagation Structures

In this section we define $\mathcal{LUP}(\phi, \mathcal{A})$, and a method for constructing it.

Definition 7 ($\mathcal{LUP}(\phi, \mathcal{A})$). *Let* Units *denote the set of unit clauses that appears during the execution of UP on* $\mathrm{CNF}(\mathrm{NaiveGnd}_{\mathcal{A}}(\phi))$. *The LUP structure for ϕ and $\mathcal{A}$ is the unique bound structure for ϕ and $\mathcal{A}$ for which:*

$$\chi^{\mathcal{A}}_{\lceil\psi\rceil}(\bar{a}) = \begin{cases} \top & \lceil\psi\rceil(\tilde{\bar{a}}) \in \text{Units} \\ \bot & \neg\lceil\psi\rceil(\tilde{\bar{a}}) \in \text{Units} \\ \infty & \text{o.w} \end{cases} \tag{3}$$

Since Algorithm 2 produces a grounding, according to Definition 6, for any bound structure, it produces a grounding for ϕ over $\mathcal{LUP}(\phi, \mathcal{A})$.

To construct $\mathcal{LUP}(\phi, \mathcal{A})$, we use an inductive definition obtained from ϕ. In this inductive definition, we use distinct vocabulary symbols for the sets of tuples which $\hat{\mathcal{A}}$ sets to true and false. The algorithm works based on the notion of *True (False) bounds*:

Definition 8 (Formula-Bound). *A* True (resp. False) bound *for a subformula $\psi(\bar{x})$ according to bound structure $\hat{\mathcal{A}}$ is the relation denoted by T_ψ (resp. F_ψ) such that:*

1. $\bar{a} \in T_\psi \Leftrightarrow \lceil\psi\rceil^{\hat{\mathcal{A}}}(\bar{a}) = \top$
2. $\bar{a} \in F_\psi \Leftrightarrow \lceil\psi\rceil^{\hat{\mathcal{A}}}(\bar{a}) = \bot$

Naturally, when $\lceil\psi\rceil^{\hat{\mathcal{A}}}(\bar{a}) = \infty$, $\bar{a}$ is not contained in either T_ψ or F_ψ.

The rules of the inductive definition are given in Table 1. These rules rules may be read as rules of FO(ID), the extension of classical logic with inductive definitions under the well-founded semantics [VGRS91,DT08], with free variables implicitly universally quantified. The *type* column indicates the type of the subformula, and the *rules* columns identify the rule for this subformula. Given a σ-structure $\mathcal{A}$, we may evaluate the definitions on $\mathcal{A}$, thus obtaining a set of concrete bounds for the subformulas of ϕ. The rules reflect the reasoning that UP can do. For example consider rule $(\vee_i \psi_i)$ of $\downarrow t$ for $\gamma(\bar{x}) = \psi_1(\bar{x}_1) \vee \cdots \vee \psi_N(\bar{x}_N)$, and for some $i \in \{1, \ldots, N\}$:

$$T_{\psi_i}(\bar{x}_i) \leftarrow T_\gamma(\bar{x}) \wedge \bigwedge_{j \neq i} F_{\psi_j}(\bar{x}_j).$$

This states that when a tuple $\bar{a}$ satisfies γ but falsifies all disjuncts, ψ_j, of γ except for one, namely ψ_i, then it must satisfy ψ_i. As a starting point, we know the value of the instance predicates, and we also assume that ϕ is $\mathcal{A}$-satisfiable.

Example 2. Let $\phi = \forall x\ \neg I_1(x) \vee E_1(x)$, $\sigma = \{I_1, I_2\}$, and $\mathcal{A} = \big(\{1,2,3,4\}; I_1^{\mathcal{A}} = \{1\}\big)$. The relevant rules from Table (1) are:

$$\begin{aligned} T_{\neg I_1(x) \vee E_1(x)}(x) &\leftarrow T_\phi \\ T_{I_1}(x) &\leftarrow I_1(x) \\ F_{\neg I_1(x)}(x) &\leftarrow T_{I_1}(x) \\ T_{E_1(x)}(x) &\leftarrow T_{\neg I_1(x) \vee E_1(x)}(x) \wedge F_{\neg I_1(x)}(x) \\ T_{E_1}(x) &\leftarrow T_{E_1(x)}(x) \end{aligned}$$

We find that $T_{E_1} = \{1\}$; in other words: $E_1(1)$ is true in each model of ϕ expanding $\mathcal{A}$.

Note that this inductive definition is monotone, because ϕ is in *Negation Normal Form (NNF)*.

type	$\downarrow t$ rules
$(\vee_i \psi_i)$	$T_{\psi_i}(\bar{x}_i) \leftarrow T_\gamma(\bar{x}) \wedge \bigwedge_{j \neq i} F_{\psi_j}(\bar{x}_j)$, for each i
$(\wedge_i \psi_i)$	$T_{\psi_i}(\bar{x}_i) \leftarrow T_\gamma(\bar{x})$, for each i
$\exists y\ \psi(\bar{x}, y)$	$T_\psi(\bar{x}, y) \leftarrow T_\gamma(\bar{x}) \wedge \forall y' \neq y\ \ F_\psi(\bar{x}, y')$
$\forall y\ \psi(\bar{x}, y)$	$T_\psi(\bar{x}, y) \leftarrow T_\gamma(\bar{x})$
$P(\bar{x})$	$T_P(\bar{x}) \leftarrow T_\gamma(\bar{x})$
$\neg P(\bar{x})$	$F_P(\bar{x}) \leftarrow T_\gamma(\bar{x})$

type	$\uparrow t$ rules
$(\vee_i \psi_i)$	$T_\gamma(\bar{x}) \leftarrow \bigvee_i T_{\psi_i}(\bar{x}_i)$, for each i
$(\wedge_i \psi_i)$	$T_\gamma(\bar{x}) \leftarrow \bigwedge_i T_{\psi_i}(\bar{x}_i)$, for each i
$\exists y\ \psi(\bar{x}, y)$	$T_\gamma(\bar{x}) \leftarrow \exists y\ T_\psi(\bar{x}, y)$
$\forall y\ \psi(\bar{x}, y)$	$T_\gamma(\bar{x}) \leftarrow \forall y\ T_\psi(\bar{x}, y)$
$P(\bar{x})$	$T_\gamma(\bar{x}) \leftarrow T_P(\bar{x})$
$\neg P(\bar{x})$	$T_\gamma(\bar{x}) \leftarrow F_P(\bar{x})$

type	$\downarrow f$ rules
$(\vee_i \psi_i)$	$F_{\psi_i}(\bar{x}_i) \leftarrow F_\gamma(\bar{x})$, for each i
$(\wedge_i \psi_i)$	$F_{\psi_i}(\bar{x}_i) \leftarrow F_\gamma(\bar{x}) \wedge \bigwedge_{j \neq i} T_{\psi_j}(\bar{x}_j)$, for each i
$\exists y\ \psi(\bar{x}, y)$	$F_\psi(\bar{x}, y) \leftarrow F_\gamma(\bar{x})$
$\forall y\ \psi(\bar{x}, y)$	$F_\psi(\bar{x}, y) \leftarrow F_\gamma(\bar{x}) \wedge \forall y' \neq y\ \ T_\psi(\bar{x}, y')$
$P(\bar{x})$	$F_P(\bar{x}) \leftarrow F_\gamma(\bar{x})$
$\neg P(\bar{x})$	$T_P(\bar{x}) \leftarrow F_\gamma(\bar{x})$

type	$\uparrow f$ rules
$(\vee_i \psi_i)$	$F_\gamma(\bar{x}) \leftarrow \bigwedge_i F_{\psi_i}(\bar{x}_i)$, for each i
$(\wedge_i \psi_i)$	$F_\gamma(\bar{x}) \leftarrow \bigvee_i F_{\psi_i}(\bar{x}_i)$, for each i
$\exists y\ \psi(\bar{x}, y)$	$F_\gamma(\bar{x}) \leftarrow \forall y\ F_\psi(\bar{x}, y)$
$\forall y\ \psi(\bar{x}, y)$	$F_\gamma(\bar{x}) \leftarrow \exists y\ F_\psi(\bar{x}, y)$
$P(\bar{x})$	$F_\gamma(\bar{x}) \leftarrow F_P(\bar{x})$
$\neg P(\bar{x})$	$F_\gamma(\bar{x}) \leftarrow T_P(\bar{x})$

Table 1: Rules for Bounds Computation

4.1 LUP Structure Computation

Our method for constructing $\mathcal{LUP}(\phi, \mathcal{A})$ is given in Algorithm 3. Several lines in the algorithm require explanation. In line 1, the $\downarrow f$ rules are omitted from the set of constructed rules. Because ϕ is in NNF, the $\downarrow f$ rules do not contribute any information to the set of bounds. To see this, observe that every $\downarrow f$ rule has an atom of the form $F_\gamma(\bar{x})$ in its body. Intuitively, for one of these rules to contribute a defined bound, certain information must have previously been obtained regarding bounds for its parent. It can be shown, by induction, that, in every case, the information about a bound inferred by an application of a $\downarrow f$ rule must have previously been inferred by a $\uparrow f$ rule. In line 2 of the algorithm we compute bounds using only the two sets of rules, $\downarrow t$ and $\uparrow f$. This is justified by the fact that applying $\{\uparrow t, \downarrow t, \uparrow f\}$ to a fixpoint has the same effect as applying $\{\downarrow t, \uparrow f\}$ to a fixpoint and then applying the $\uparrow t$ rules afterwards. So we postpone the execution of the $\uparrow t$ rules to line 7.

Line 3 checks for the case that the definition has no model, which is to say that the rules allow us to derive that some atom is both in the true bound and the false bound for some subformula. This happens exactly when UP applied to the naive grounding would detect inconsistency.

Finally, in lines 6 and 7 we throw away the true bounds for all non-atomic subformulas, and then compute new bounds by evaluating the $\uparrow t$ rules, taking already computed bounds (with true bounds for non-atoms set to empty) as the initial bounds in the computation. To see why, observe that the true bounds computed in line 2 are based on the assumption that ϕ is $\mathcal{A}$-satisfiable. So $\lceil \phi \rceil$ is set to true which stops the top-down bounded grounding algorithm of Section 3.2 from producing a grounding for ϕ. That is because the $Simpl$ function, considering the true bound for the ϕ, simply returns $\top$ instead of calling $Gnd_{\hat{\mathcal{A}}}(.,.)$ on subformulas of the ϕ. This also holds for all the formulas with true-bounds, calculated this way, except for the atomic formulas. So, we delete these true bounds based on the initial unjustified assumption, and

Algorithm 3 Computation of $\mathcal{LUP}(\phi, \mathcal{A})$

1: Construct the rules $\{\uparrow t, \downarrow t, \uparrow f\}$
2: Compute bounds by evaluating the inductive definition $\{\downarrow t, \uparrow f\}$
3: **if** Bounds are inconsistent **then**
4: **return** "$\mathcal{A}$ has no solution"
5: **end if**
6: Throw away $T_\psi(\bar{x})$ for all non-atomic subformulas $\psi(\bar{x})$
7: Compute new bounds by evaluating the inductive definition $\{\uparrow t\}$
8: **return** LUP structure constructed from the computed bounds, according to Definition 8 .

then construct the correct true bounds by application of the $\uparrow t$ rules, in line 7. This is the main reason for postponing the execution of $\uparrow t$ rules.

5 Bottom-up Grounding over Bound Structures

The grounding algorithm we use in Enfragmo constructs a grounding by a bottom-up process that parallels database query evaluation, based on an extension of the relational algebra. We give a rough sketch of the method here: further details can be found in, e.g., [Moh04,PLTG07]. Given a structure (database) $\mathcal{A}$, a boolean query is a formula ϕ over the vocabulary of $\mathcal{A}$, and query answering is evaluating whether ϕ is true, i.e., $\mathcal{A} \models \phi$. In the context of grounding, ϕ has some additional vocabulary beyond that of $\mathcal{A}$, and producing a reduced grounding involves evaluating out the instance vocabulary, and producing a ground formula representing the expansions of $\mathcal{A}$ for which ϕ is true.

For each sub-formula $\alpha(\bar{x})$ with free variables $\bar{x}$, we call the set of reduced groundings for α under all possible ground instantiations of $\bar{x}$ an answer to $\alpha(\bar{x})$. We represent answers with tables on which the extended algebra operates. An X-relation, in databases, is a k-ary relation associated with a k-tuple of variables X, representing a set of instantiations of the variables of X. Our grounding method uses extended X-relations, in which each tuple $\bar{a}$ is associated with a formula. In particular, if R is the answer to $\alpha(\bar{x})$, then R consists of the pairs $(\bar{a}, \alpha(\tilde{\bar{a}}))$. Since a sentence has no free variables, the answer to a sentence ϕ is a zero-ary extended X-relation, containing a single pair $(\langle\rangle, \psi)$, associating the empty tuple with formula ψ, which is a reduced grounding of ϕ.

The relational algebra has operations corresponding to each connective and quantifier in FO: complement (negation); join (conjunction); union (disjunction), projection (existential quantification); division or quotient (universal quantification). Each generalizes to extended X-relations. If $(\bar{a}, \alpha(\tilde{\bar{a}})) \in \mathcal{R}$ then we write $\delta_{\mathcal{R}}(\bar{a}) = \alpha(\tilde{\bar{a}})$. For example, the join of extended X-relation $\mathcal{R}$ and extended Y-relation $\mathcal{S}$ (both over domain A), denoted $\mathcal{R} \bowtie \mathcal{S}$, is the extended $X \cup Y$-relation $\{(\bar{a}, \psi) \mid \bar{a} : X \cup Y \to A, \bar{a}|_X \in \mathcal{R}, \bar{a}|_Y \in \mathcal{S}, \text{ and } \psi = \delta_{\mathcal{R}}(\bar{a}|_X) \wedge \delta_{\mathcal{S}}(\bar{a}|_Y)\}$; It is easy to show that, if $\mathcal{R}$ is an answer to $\alpha_1(\bar{x})$ and $\mathcal{S}$ is an answer to $\alpha_2(\bar{y})$ (both wrt $\mathcal{A}$), then $\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\alpha_1(\bar{x}) \wedge \alpha_2(\bar{y})$. The analogous property holds for the other operators.

To ground with this algebra, we define the answer to atomic formula $P(\bar{x})$ as follows. If P is an instance predicate, the answer is the set of tuples $(\bar{a}, \top)$, for $\bar{a} \in P^{\mathcal{A}}$. If P is an expansion predicate, the answer is the set of all tuples $(\bar{a}, P(\bar{a}))$, for $\bar{a}$ a tuple of elements from the domain of $\mathcal{A}$. Then we apply the algebra inductively, bottom-up, on the structure of the formula. At the top, we obtain the answer to ϕ, which is a relation containing only the pair $(\langle\rangle, \psi)$, where ψ is a reduced grounding of ϕ wrt $\mathcal{A}$.

Example 3. Let $\sigma = \{P\}$ and $\varepsilon = \{E\}$, and let $\mathcal{A}$ be a σ-structure with $P^{\mathcal{A}} = \{(1,2,3),(3,4,5)\}$. The following extended relation $\mathcal{R}$ is an answer to $\phi_1 \equiv P(x,y,z) \wedge E(x,y) \wedge E(y,z)$:

x	y	z	ψ
1	2	3	$E(1,2) \wedge E(2,3)$
3	4	5	$E(3,4) \wedge E(4,5)$

Observe that $\delta_{\mathcal{R}}(1,2,3) = E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi_1[(1,2,3)] = P(1,2,3) \wedge E(1,2) \wedge E(2,3)$, and $\delta_{\mathcal{R}}(1,1,1) = \bot$ is a reduced grounding of $\phi_1[(1,1,1)]$.

The following extended relation is an answer to $\phi_2 \equiv \exists z \phi_1$:

x	y	ψ
1	2	$E(1,2) \wedge E(2,3)$
3	4	$E(3,4) \wedge E(4,5)$

Here, $E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi_2[(1,2)]$. Finally, the following represents an answer to $\phi_3 \equiv \exists x \exists y \phi_2$, where the single formula is a reduced grounding of ϕ_3.

ψ
$[E(1,2) \wedge E(2,3)] \vee [E(3,4) \wedge E(4,5)]$

To modify the algorithm to ground using $\mathcal{LUP}(\phi, \mathcal{A})$ we need only change the base case for expansion predicates. To be precise, if P is an expansion predicate we set the answer to $P(\bar{x})$ to the set of pairs $(\bar{a}, \psi)$ such that:

$$\psi = \begin{cases} P(\tilde{\bar{a}}) & \text{if } P^{\mathcal{LUP}(\phi,\mathcal{A})}(\bar{a}) = \infty \\ \top & \text{if } P^{\mathcal{LUP}(\phi,\mathcal{A})}(\bar{a}) = \top \\ \bot & \text{if } P^{\mathcal{LUP}(\phi,\mathcal{A})}(\bar{a}) = \bot. \end{cases}$$

Observe that bottom-up grounding mimics the second phase of Algorithm 3, i.e., a bottom-up truth propagation, except that it also propagates the falses. So, for bottom up grounding, we can omit line 7 from Algorithm 3.

Proposition 2. *Let $(\langle\rangle, \psi)$ be the answer to sentence ϕ wrt $\mathcal{A}$ after LUP initialization, then:*

$$Gnd_{\mathcal{LUP}(\phi,\mathcal{A})}(\phi, \emptyset) \equiv \psi$$

where $Gnd_{\mathcal{LUP}(\phi,\mathcal{A})}(\phi, \emptyset)$ is the result of top-down grounding Algorithm 2 of ϕ over LUP structure $\mathcal{LUP}(\phi, \mathcal{A})$.

This bottom-up method uses only the reduct of $\mathcal{LUP}(\phi, \mathcal{A})$ defined by $\sigma \cup \varepsilon \cup \tilde{A}$, not the entire LUP structure.

6 Experimental Evaluation of LUP

In this section we present an empirical study of the effect of LUP on grounding size and on grounding and solving times. We also compare LUP with GWB in terms of these same measures. The implementation of LUP is within our bottom-up grounder Enfragmo, as described in this paper, and the implementation of GWB is in the top-down grounder GIDL, which is described in [WMD08b,WMD10]. GIDL has several parameters to control the precision of the bounds computation. In our experiments we use the default settings. We used MINISAT as the ground solver for Enfragmo. GIDL produces an output specifically for the ground solver MINISAT(ID), and together they form the IDP system [WMD08d].

We report data for instances of three problems: Latin Square Completion, Bounded Spanning Tree and Sudoku. The instances are latin_square.17068* instances of Normal Latin Square Completion, the 104_rand_45_250_* and 104_rand_35_250_* instances of BST, and the ASP contest 2009 instances of Sudoku from the Asparagus repository[3]. All experiments were run on a Dell Precision T3400 computer with a quad-core 2.66GHz Intel Core 2 processor having 4MB cache and 8GB of RAM, running CentOS 5.5 with Linux kernel 2.6.18.

In Tables 2 and 4, columns headed "Literals" or "Clauses" give the number of literals or clauses in the CNF formula produced by Enfragmo without LUP (our baseline), or these values for other grounding methods expressed as a percentage of the baseline value. In Tables 3 and 5, all values are times seconds. All values give are means for the entire collection of instances. Variances are not given, because they are very small. We split the instances of BST, into two sets, based on the number of nodes (35 or 45), because these two groups exhibit somewhat different behaviour, but within the groups variances are also small. In all tables, the minimum (best) values for each row are in bold face type, to highlight the conditions which gave best performance.

Table 2 compares the sizes of CNF formulas produced by Enfragmo without LUP (the base line) with the formulas obtained by running UP on the baseline formulas and by running Enfragmo with LUP. Clearly LUP reduces the size at least as much as UP, and usually reduces the size much more, due to the removal of autarkies.

Total time for solving a problem instance is composed of grounding time and SAT solving time. Table 3 compares the grounding and SAT solving time with and without LUP bounds. It is evident that the SAT solving time is always reduced with LUP. This reduction is due to the elimination of the unit clauses and autark subformulas from the grounding. Autark subformula elimination also affects the time required to convert the ground formula to CNF which reduces the grounding time, but in some cases the overhead

[3] http://asparagus.cs.uni-potsdam.de

	Enfragmo		Enfragmo+UP (%)		Enfragmo+LUP (%)	
Problem	Literals	Clauses	Literals	Clauses	Literals	Clauses
Latin Square	7452400	2514100	**0.07**	**0.07**	**0.07**	**0.07**
BST 45	22924989	9061818	0.96	0.96	**0.24**	**0.24**
BST 35	8662215	3415697	0.95	0.96	**0.37**	**0.37**
Sudoku	2875122	981668	0.17	0.18	**0.07**	**0.08**

Table 2: Impact of LUP on the size of the grounding. The first two columns give the numbers of literals and clauses in groundings produced by Enfragmo without LUP (the baseline). The other columns give these measures for formulas produced by executing UP on the baseline groundings (Enfragmo+UP), and for groundings produced by Enfragmo with LUP (Enfragmo+LUP), expressed as a fraction baseline values.

	Enfragmo			Enfragmo with LUP			Speed Up Factor		
Problem	Gnd	Solving	Total	Gnd	Solving	Total	Gnd	Solving	Total
Latin Square	**0.89**	1.39	**2.28**	3.27	**0.34**	3.61	-2.38	1.05	-1.33
BST 45	6.08	7.56	13.64	**2**	**1.74**	**3.74**	4.07	5.82	9.9
BST 35	2.13	2.14	4.27	**1.07**	**0.46**	**1.53**	1.06	1.68	2.74
Sudoku	**0.46**	1.12	**1.59**	2.08	**0.26**	2.34	-1.62	0.86	-0.76

Table 3: Impact of LUP on reduction in both grounding and (SAT) solving time. Grounding time here includes LUP computations and CNF generation.

imposed by LUP computation may not be made up for by this reduction. As the table shows, when LUP outperforms the normal grounding we get factor of 3 speed-ups, whereas when it loses to normal grounding the slowdown is by a factor of 1.5.

Table 4 compares the size reductions obtained by LUP and by GWB in GIDL. The output of GIDL contains clauses and rules. The rules are transformed to clauses in (MINISAT(ID)). The measures reported here are after that transformation. LUP reduces the size much more than GWB, in most of the cases. This stems from the fact that GIDL's bound computation does not aim for completeness wrt unit propagation. This also affects the solving time because the CNF formulas are much smaller with LUP as shown in Table 5. Table 5 shows that Enfragmo with LUP and MINISAT is always faster than GIDL with MINISAT(ID) with or without bounds, and it is in some cases faster than Enfragmo without LUP.

7 Discussion

In the context of grounding-based problem solving, we have described a method we call lifted unit propagation (LUP) for carrying out a process essentially equivalent to unit propagation before and during grounding. Our experiments indicate that the method can substantially reduce grounding size – even more than unit propagation itself, and sometimes reduce total solving time as well.

	Enfragmo (no LUP)		GIDL (no bounds)		Enfragmo with LUP		GIDL with bounds	
Problem	Literals	Clauses	Literals	Clauses	Literals	Clauses	Literals	Clauses
Latin Square	7452400	2514100	0.74	0.84	**0.07**	**0.07**	0.59	0.61
BST 45	22924989	9061818	0.99	1.02	**0.24**	**0.24**	0.25	**0.24**
BST 35	8662215	3415697	1.01	1.04	**0.37**	**0.37**	0.39	0.39
Sudoku	2875122	981668	0.56	0.6	**0.07**	**0.08**	0.38	0.39

Table 4: Comparison between the effectiveness of LUP and GIDL Bounds on reduction in grounding size. The columns under Enfragmo show the actual grounding size whereas the other columns show the ratio of the grounding size relative to that of Enfragmo (without LUP).

	Enfragmo			IDP			Enfragmo+LUP			IDP (Bounds)		
Problem	Gnd	Solving	Total	Gnd	Solving	Total	Gnd	Solving	Total	Gnd	Solving	Total
Latin Square	**0.89**	1.39	**2.28**	3	4.63	7.63	3.27	**0.34**	3.61	2.4	3.81	6.21
BST 45	6.08	7.56	13.64	7.25	20.84	28.09	2	**1.74**	**3.74**	**1.14**	4.45	5.59
BST 35	2.13	2.14	4.27	2.63	6.31	8.94	1.07	**0.46**	**1.53**	**0.67**	2.73	3.4
Sudoku	**0.46**	1.12	**1.59**	1.81	1.3	3.11	2.08	**0.26**	2.34	2.85	0.51	2.37

Table 5: Comparison of solving time for Enfragmo and IDP, with and without LUP/bounds.

Our work was motivated by the results of [WMD08b,WMD10], which presented the method we have referred to as GWB. In GWB, bounds on sub-formulas of the specification formula are computed without reference to an instance structure, and represented with FO formulas. The grounding algorithm evaluates instantiations of these bound formulas on the instance structure to determine that certain parts of the naive grounding may be left out. If the bound formulas exactly represent the information unit propagation can derive, then LUP and GWB are equivalent (though implemented differently). However, generally the GWB bounds are weaker than the LUP bounds, for two reasons. First, they must be weaker, because no FO formula can define the bounds obtainable with respect to an arbitrary instance structure. Second, to make the implementation in GIDL efficient, the computation of the bounds is heuristically truncated. This led us to ask how much additional reduction in formula size might be obtained by the complete LUP method, and whether the LUP computation could be done fast enough for this extra reduction to be useful in practice.

Our experiments with the Enfragmo and GIDL grounders show that, at least for some kinds of problems and instances, using LUP can produce much smaller groundings than the GWB implementation in GIDL. In our experiments, the total solving times for Enfragmo with ground solver MINISAT were always less than those of GIDL with ground solver MINISAT(ID). However, LUP reduced total solving time of Enfragmo with MINISAT significantly in some cases, and increased it — albeit less significantly — in others. Since there are many possible improvements of the LUP implementation, the question of whether LUP can be implemented efficiently enough to be used all the time remains unanswered.

Investigating more efficient ways to do LUP, such as by using better data structures, is a subject for future work, as is consideration of other approximate methods such, as placing a heuristic time-out on the LUP structure computation, or dovetailing of the LUP computation with grounding. We also observed that the much of the reduction in grounding size obtained by LUP is due to identification of autark sub-formulas. These cannot be eliminated from the naive grounding by unit propagation. Further investigation of the importance of these in practice is another direction we are pursuing. One more direction we are pursuing is the study of methods for deriving even stronger information that represented by the LUP structure, to further reduce ground formula size, and possibly grounding time as well.

Acknowledgements

The authors are grateful to Marc Denecker, Johan Wittocx, Bill MacReady, Calvin Tang, Amir Aavani, Shahab Tasharoffi, Newman Wu, D-Wave Systems, MITACS, and NSERC.

References

[ATÜ+10] Amir Aavani, Shahab Tasharrofi, Gulay Ünel, Eugenia Ternovska, and David G. Mitchell. Speed-up techniques for negation in grounding. In *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 13–26. Springer, 2010.

[AWTM11] Amir Aavani, Xiongnan (Newman) Wu, Eugenia Ternovska, and David G. Mitchell. Grounding formulas with complex terms. In *Canadian AI*, volume 6657 of *LNCS*, pages 13–25. Springer, 2011.

[DT08] Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Logic*, 9:14:1–14:52, April 2008.

[DW08] M. Denecker and J. Wittocx. Personal communication, 2008.

[ET06] D. East and M. Truszczynski. Predicate-calculus based logics for modeling and solving search problems. *ACM Trans. Comput. Logic (TOCL)*, 7(1):38 – 83, 2006.

[GKK+08] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Unpublished draft, 2008.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[Moh04] Raheleh Mohebali. A method for solving NP search based on model expansion and grounding. Master's thesis, Simon Fraser University, 2004.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Appl. Math.*, 10:287–295, March 1985.

[MT11] David Mitchell and Eugenia Ternovska. Knowledge representation, search problems and model expansion. In *Knowing, Reasoning and Acting: Essays in Honour of Hector J. Levesque*, pages 347–362, 2011.

[MTHM06] David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, December 2006.

[PLTG07] M. Patterson, Y. Liu, E. Ternovska, and A. Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *Proc. IJCAI'07)*, 2007.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[Tse68] G.S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115 – 125, 1968.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[WMD08a] Johan Wittocx, Maarten Mariën, and Marc Denecker. GidL: A grounder for FO+. In *In Proc., Twelfth International Workshop on NMR,*, pages 189–198, September 2008.

[WMD08b] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding with bounds. In *AAAI*, pages 572–577, 2008.

[WMD08c] Johan Wittocx, Maarten Marien, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *Proc., LaSh 2008*, 2008.

[WMD08d] Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.

[WMD10] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR)*, 38:223–269, 2010.