TLP **13** (4–5): Online Supplement, July 2013.

URL: http://dx.doi.org/10.1017/S1471068413000410

SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support

PAULA-ANDRA BUSONIU[1], JOHANNES OETSCH[1], JÖRG PÜHRER[1], PETER SKOČOVSKÝ[2] and HANS TOMPITS[1]*

[1]*Technische Universität Wien, Institut für Informationssysteme 184/3, Favoritenstrasse 9-11, A-1040 Vienna, Austria*

[2]*Universidade Nova de Lisboa, CENTRIA and Departamento de Informatica, 2829-516 Caparica, Portugal*

(*e-mail:* {andra.busoniu,aifargonos}@gmail.com)

(*e-mail:* {oetsch,puehrer,tompits}@kr.tuwien.ac.at)

submitted 10 April 2013; revised 23 June 2013; accepted 5 July 2013

Abstract

In this paper, we present SeaLion, an integrated development environment (IDE) for answer-set programming (ASP). SeaLion provides source-code editors for the languages of Gringo and DLV and offers popular amenities like syntax highlighting, syntax checking, code completion, visual program outline, and refactoring functionality. The tool has been realised in the context of a research project whose goal is the development of techniques to support the practical coding process of answer-set programs. In this respect, SeaLion is the first IDE for ASP that provides debugging features that work for real-world answer-set programs and supports the rich languages of modern answer-set solvers. Indeed, SeaLion implements a stepping-based debugging approach that allows the developer to quickly track down programming errors by simply following his or her intuitions on the intended semantics. Besides that, SeaLion supports ASP development using model-driven engineering techniques including domain modelling with extended UML class diagrams and visualisation of answer sets in corresponding instance diagrams. Moreover, customised visualisation as well as visual editing of answer sets is realised by the Kara plugin of SeaLion. Further implemented features are a documentation generator based on the LANA annotation language, support for external solvers, and interoperability with external tools. SeaLion comes as a plugin of the popular Eclipse platform and provides interfaces for future extensions of the IDE.

1 Introduction

Answer-set programming (ASP) is a well-established paradigm for declarative problem solving (Gelfond and Leone 2002) that has its roots in nonmonotonic reasoning, knowledge representation, and logic programming. The main idea of ASP is to represent solutions to computational problems in terms of logic programs such that the stable models (Gelfond and Lifschitz 1988) of the latter, referred to as

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

their *answer sets*, provide the solutions of a problem instance. In recent years, the expressibility of languages supported by answer-set solvers increased significantly (Gebser *et al.* 2009). Also, ASP solvers have become much more efficient; e.g., the solver `Clasp` proved to be competitive with state-of-the-art SAT solvers (Le Berre *et al.* 2009; Järvisalo *et al.* 2012). Despite these improvements in solver technology, a lack of suitable *engineering tools* for ASP led to the launch of a project on methods and methodologies for developing answer-set programs (Oetsch *et al.* 2010), whose goal is not only research into new support techniques for programmers but also their realisation in an integrated development environment (IDE) which we present in this paper. The system is called `SeaLion`, where "Sea" stands for *Support Environment for ASP*, and the Lion symbolises the strength and good-naturedness that we aimed to integrate into the environment. It is designed as a plugin of the *Eclipse platform* (Eclipse Project 2013) and offers functionality like source editors, syntax highlighting, syntax checking, visual program outline, and refactoring for the languages of the state-of-the-art solvers `Clasp` (in conjunction with `Gringo`) (Gebser *et al.* 2007, 2009) and `DLV` (Leone *et al.* 2006).

A preliminary report on `SeaLion` (Oetsch *et al.* 2011) discussed initial functionality and an outline on planned features of the system. Most of these plans have been realised in the meanwhile. Above all, `SeaLion` is now equipped with a debugging framework that can cope with real-world answer-set programs—a feature that has been requested from the ASP community for a long time. Indeed, `SeaLion` implements a stepping-based debugging approach (Oetsch *et al.* 2011, 2013) that allows the developer to quickly track down errors by following his or her intuitions on the intended semantics. The technique is intuitive and similar in spirit as stepwise debugging in imperative languages but also respects the declarative nature of the answer-set semantics. Another debugging approach (Oetsch *et al.* 2010; Polleres *et al.* 2013) has recently been realised as a `SeaLion` plugin (Frühstück *et al.* 2013) called `Ouroboros`. It tackles the question why a given interpretation is not an answer set. While it provides additional debugging functionality for `SeaLion`, `Ouroboros` also profits from the stepping plugin which can help in building up the interpretation in question. Another simple yet handy feature of `SeaLion` is the search for a rule that derived a particular atom in a computed answer set.

The source editors of `SeaLion` allow for augmenting answer-set programs with annotations written in Lana (Language for ANnotating Answer-set programs), an annotation language for structuring, documenting, and testing answer-set programs (De Vos *et al.* 2012). These annotations allow for adding valuable information to a program that can be exploited by various tools, including many features of `SeaLion`. For instance, `SeaLion` integrates the features of the documentation generator `ASPDoc` that can generate HTML documentation of answer-set programs.

Lana is also used by `SeaLion`'s modelling framework that adopts techniques from *model-driven engineering* (MDE) (Schmidt 2006) for supporting the development of answer-set programs. In object-oriented programming, it is common to model the data structures needed by means of graphical models like *UML class diagrams* (Fowler 2004). These *domain models* serve as primary development artefacts from which parts of the code can be generated. We implemented a graphical editing

framework for modelling the domain of an answer-set program in an extended UML class diagram. The model can then be translated into an ASP predicate scheme, stored in ASP source files using LANA annotations. The translation includes documentation and assertions that allow, e.g., to check violation of modelled constraints. Once a domain model is created, answer sets can be visualised in *UML object diagrams* that display the instances of the classes defined in the model that are encoded in the answer set and their relations.

Visualisation of answer sets is provided by the `Kara` plugin of `SeaLion` (Kloimüllner *et al.* 2011) that can create user-defined graphical representations of interpretations. Morevover, `Kara` offers generic visualisations and allows for graphically manipulating interpretations.

Related to this work, also other IDEs for ASP were developed in recent years., viz. `ASPIDE` (Febbraro *et al.* 2011), `APE` (Sureshkumar *et al.* 2007), and `iGROM` (Koziarkiewicz 2011).

The remainder of the paper is organised as follows. In the next section, we discuss design choices and implementation principles we followed as well as how to obtain `SeaLion`. Section 3 gives an overview of the main features of `SeaLion`. In Section 4, we discuss other systems related to `SeaLion`. Section 5 concludes the paper with an outline of future work.

2 Implementation and availability

The target audience for `SeaLion` are software developers new to ASP yet familiar with support tools as used in procedural and object-oriented programming. As a consequence, it was our aim to create an environment that is similiar to well-established development tools. In particular, this was one reason why `SeaLion` is implemented as plugin of the Eclipse platform which is popular among software engineers and can be considered the standard environment for Java development. Arguably, people who are familiar with Eclipse and basic ASP skills will easily adapt to `SeaLion`. The decision to build on Eclipse rather than writing a stand-alone application from scratch has further 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 highlighting, problem reporting, project management, the undo-redo mechanism, the console view, the refactoring and the navigation frameworks (Outline), 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), as well as task management.

A key aspect in the design of `SeaLion` is extensibility. That is, the API framework is tailored to support, on the one hand, further ASP languages with low effort and, on the other hand, allows for embedding future features easily. The `SeaLion` implementation follows itself a modular principle, where different features are Eclipse plugins themselves.

Regarding the user interface, our aim was to make the usage of SeaLion as smooth as possible. We paid attention that the methods of our features can be performed with as few mouse clicks or other user interaction as necessary and followed Eclipse conventions for shortcuts. Moreover, we wanted to give the ASP developer much freedom in how to use the system. For example, SeaLion avoids functionality that patronises the ASP developer like imposing a certain coding style, i.e., every valid ASP source file should be usable in our IDE. Furthermore, we aimed at interoperability, e.g., through the use of standards or the framework for external tool configurations that allows for using arbitrary external tools, e.g., for postprocessing computed answer sets.

SeaLion is free software published under the GNU General Public License version 3. There are two major options to install SeaLion. Users of Eclipse can obtain it using Eclipse's update mechanism with the benefit of automatic updates. Alternatively, we provide standalone packages of SeaLion for different operating systems and architectures that only require a Java Runtime. For both installation variants, we provide SeaLion packages with pre-configured ASP solvers. For more information on SeaLion, installation instructions, and links to the source code, we refer to the project web site

http://www.sealion.at.

3 Main features

In this section, we give an overview of the main features of SeaLion and thereby focus on recent key features that were not yet covered in an earlier preliminary report on SeaLion (Oetsch *et al.* 2011). In particular, after describing the basic IDE functionality in Section 3.1, we will concentrate on the stepping feature in Section 3.2. Finally, we describe our model-driven engineering environment and the Kara plugin for visualisation and visual editing of interpretations in Sections 3.3 and 3.4, respectively.

3.1 Basic functionality

The central element in SeaLion is the *source-code editor* for logic programs. Altough there are current endeavours to harmonise solver languages (cf. also Section 5), up to now the languages of Gringo and DLV differ in their presentation of aggregates and many other small details. That is why the SeaLion editor comes in two variants, one for DLV and one for Gringo. A screenshot of a Gringo source file in SeaLion's editor is given in Figure 1.

The editors provide typical conveniences of IDEs, like context-sensitive *syntax highlighting*, *syntax checking*, and *problem reporting*. Terms and predicates appearing in the program are proposed for *autocompletion*. SeaLion also offers functionality for *refactoring* answer-set programs. In particular, we implemented functionality for uniform and safe renaming of predicates, constants, function symbols, and variables throughout a user-defined set of files containing answer-set programs. Once a new

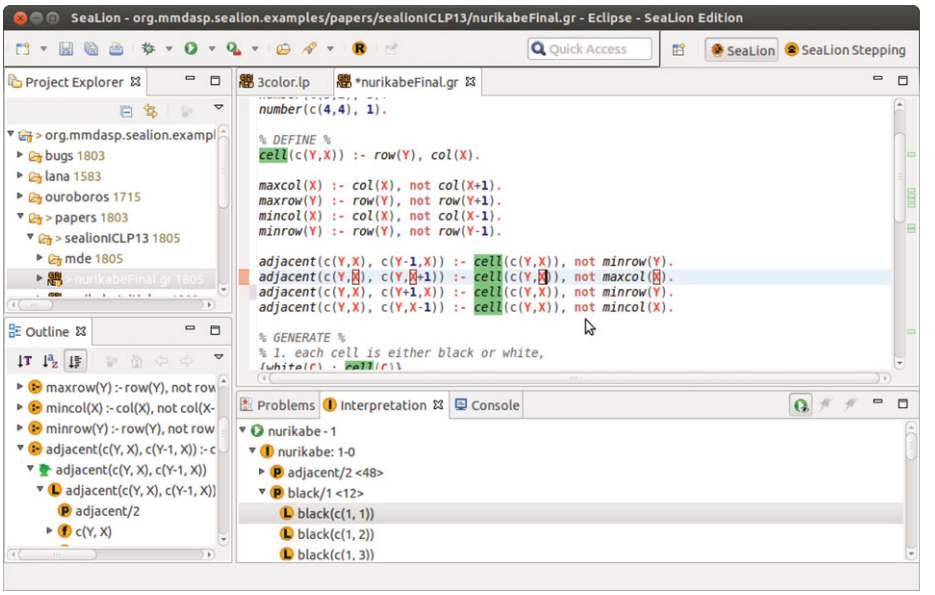

Fig. 1. (Colour online) A screenshot of SeaLion's editor, the program outline, and the interpretation view.

name is chosen, the user has the possibility to directly apply the changes implied by renaming or revise them on a preview page. Here, one can inspect the effects file by file where the original as well as the new source code are displayed next to each other and all hypothetical changes are highlighted as depicted in Figure B1 in the online appendix of this paper. An overview of the edited answer-set program is given in Eclipse's *Outline View* in a tree-shaped graphical representation that can be seen in the bottom-right corner of Figure 1. Clicking on a node of the tree selects the source code corresponding to the represented program element in the editor such that the programmer can proceed editing there. Another convenient editor feature is the temporary highlighting of code the programmer might be interested in. For instance, if the cursor is positioned over a literal, the positions of all literals of the same predicate in the overall document are indicated.

The editors are capable of processing LANA annotations (De Vos *et al.* 2012) that allow for documenting code, structuring ASP programs by grouping rules into coherent blocks, and specifying, e.g., language signatures, assertions, as well as unit tests for such blocks. Annotations are invisible to an ASP solver since they have the form of program comments, but they can be interpreted by specialised support tools. Also, SeaLion exploits LANA annotations. For instance, LANA descriptions of terms and predicates appear next to autocompletion proposals. Moreover, SeaLion allows for automatically generating source code documentation for answer-set programs, similar as tools like JavaDoc or Doxygen do for other programming languages based on LANA annotations. For this purpose, the IDE incorporates the ASPDoc documentation generator that takes LANA-annotated ASP code as input and produces HTML files as output (De Vos *et al.* 2012). The documentation contains

descriptions of all blocks of the answer-set program, where sub-blocks are indented with respect to their parent blocks. Also, a summary of the block structure of the entire answer-set program is presented at the beginning of the documentation to provide an overview. For each block, descriptions of the used atoms and types of involved terms, as well as for assertions, are given. The documentation also includes HTML versions of the program's source code, which can be particularly useful for sharing ASP code online. There are links from the documentation to the source code and vice versa. Likewise, rules for defining pre- and postconditions can be inspected by using respective links. ASP documentation generation can be accessed through Eclipse's export menu.

In order to interact with solvers, grounders, and other ASP-related tools, `SeaLion` has 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`. On our website, we offer `SeaLion` packages that include or automatically install a variety of popular grounders and solvers for which external tool configurations are pre-defined. 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 example, one can define two separate tool configurations for `Gringo` and `Clasp` and define a piped tool configuration 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. As arbitrary tools can be piped, this mechanism allows for post-processing or handling solver output as needed, e.g., opening external visualisation tools like `IDPDraw` (Wittocx 2009) and `ASPVIZ` (Cliffe *et al.* 2008). Default solvers for different solver languages can be set in the preference menu of `SeaLion` depending on file content types in the "Content Type Preferences" section.

For executing answer-set solvers and other tools, we make use of Eclipse's *launch configuration framework*, i.e., the user can create re-usable *launch configurations* that define which programs should be executed using particular external-tool configurations, the command-line arguments to use, and other settings. Figure B3 shows the page of the launch configuration editor on which input files for a solver invocation can be selected. Moreover, a launch configuration contains information how the output of the solver should be treated. One option is to print the solver output as it is in Eclipse's *console view*. The other option is to parse the resulting answer sets for further use in `SeaLion`. In this case, the answer sets obtained from the solvers are stored in `SeaLion`'s *interpretation view* as well as in the *interpretation comparison view*. In both, interpretations are visualised as expandable trees, where the literals of each interpretation are grouped by their predicates. Compared to a standard textual representation, this way of visualising answer sets provides a well-arranged overview of the individual interpretations. While the interpretation view lists interpretations in rows, the interpretation comparison view places them in columns. By horizontally arranging trees for different interpretations next to each other, it is easy to compare two or more interpretations.

Besides defining launch configurations, `SeaLion` also offers the possibility to invoke a solver right away on a selection of files in the workspace using the default settings of an external tool configuration. This is realised using the so-called *Launch Shortcuts* mechanism of Eclipse. The user selects the files that should be evaluated in the project explorer and selects the `SeaLion` entry of their "Run As" context menu. The entry is available as soon as an external tool configuration is set as default solver for the selected file content type.

3.2 Stepping

Next, we discuss the stepping feature of `SeaLion`, being its primary debugging mechanism. Stepping for ASP was introduced as a strategy to identify mismatches between the intended semantics of an answer-set program under development and its actual semantics (Oetsch *et al.* 2011). The general idea is to monotonically build up an interpretation by adding, in each step, literals derived by a rule that is active with respect to the interpretation obtained in the previous step (a rule is active under an interpretation if that interpretation satisfies the rule's body). The process is interactive in the sense that at each such step the user chooses the active rule to proceed with and decides which literals of the rule should be considered true or false in the target interpretation. The computation model used in stepping ensures that, if the interpretation specified in this way is indeed an answer set, the process of stepping will eventually terminate with the interpretation as its result. Otherwise, it will get stuck at some step and the user gets insight why the interpretation is not an answer set, e.g., when a constraint becomes irrevocably active. A computation that will inevitably get stuck is called *failed*. Due to the declarativity of ASP, once one detects unintended semantics, it can be a tough problem to manually detect the reason. Stepping is a method for breaking this problem into smaller parts and structuring the search for an error. At the same time, relying on the user's intuition on which rules to proceed with, stepping can be guided such that the search quickly results in new insights. The approach is inspired by stepping-based debugging for procedural languages, where the behaviour of a program is analysed by executing statement by statement, following the program's control flow, and inspecting variable assignments. It turns out that declarativity in ASP is not in discrepancy with adapting a method from the imperative paradigm, but fruitful instead. That is, on the one hand, with stepping the user always has guidance for starting the search for bugs, and, on the other hand, the interactive choice for the next rule makes stepping in ASP in a sense more flexible than traditional stepping, where the control flow dictates which statements are to be considered next. To still allow for fast debugging, procedural language debuggers allow to set *breakpoints*, i.e., to mark statements until which execution is done automatically. We also have a similar feature in stepping for ASP, called *jumping*, that allows to consider multiple rules which are assumed to be correct at once. Hence, we can speed up stepping by only inspecting suspicious parts of the program step by step.

The original ASP stepping framework was introduced for normal logic programs (Oetsch *et al.* 2011). Therefore, it has merely been of theoretical interest as typically

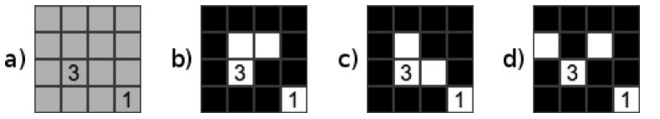

Fig. 2. Nurikabe examples: a) problem instance, b) correct solution, c) wrong solution violating (iii), d) wrong solution violating (v).

answer-set programs involve aggregates, conditions, and other language features provided by modern solvers. Moreover, the use of function symbols in answer-set programs leads to infinite programs obtained by the naïve grounding that was assumed in the previous approach. To overcome these major limitations, we lifted the stepping framework to DL-programs (Oetsch *et al.* 2012), and finally to the languages of `Gringo` and `DLV` (Oetsch *et al.* 2013). Hence, we could implement stepping in `SeaLion` fully covering these two ASP languages. Note that a formal introduction of the stepping framework is beyond the scope of this system paper and will be reported in a companion paper (Oetsch *et al.* 2013).

We now explain the realisation of stepping in `SeaLion` using the following example.

Example 1

Suppose user Leo writes a program in `SeaLion` that solves a Nurikabe puzzle which is a grid of cells, some of which contain natural numbers.[1] The goal is to colour the cells such that (i) each cell is either black or white, (ii) cells with numbers are white, (iii) there are no 2x2 blocks of black cells, (iv) all black cells are transitively connected, and (v) each maximal group of white cells that are transitively connected must contain exactly one cell with a number (this number must be the number of cells in the group), where two cells are considered connected if they are of the same colour and share a border. An example of a puzzle instance, its solution, and two wrong solutions, can be seen in Figure 2.

The input expected by Leo's program consists of facts using predicates `row/1` and `col/1` that enumerate row and column numbers and atoms of form `number(c(Y,X),N)`, indicating that the cell `c(Y,X)` contains number `N`. The program, whose full source code is given in the online appendix, consists of three parts: a generate part that guesses the colour for each cell, a define part specifying when two cells are orthogonal adjacent, and a check part that has auxiliary definitions of reachability and cell groups, implementing conditions (ii)–(v) as constraints.

Generally, a bug is detected when, for a particular input of a program, actual output differs from the intended output. For ASP, there may be two cases: (1) an intended answer set is missing from the actual output, or (2) the actual output contains an unintended answer set.

Example 2

When Leo runs the program from our running example for input

```
row(1..4). col(1..4). number(c(3,2),3). number(c(4,4),1).
```

[1] Similar puzzles can be found at `http://www.puzzle-nurikabe.com/`.

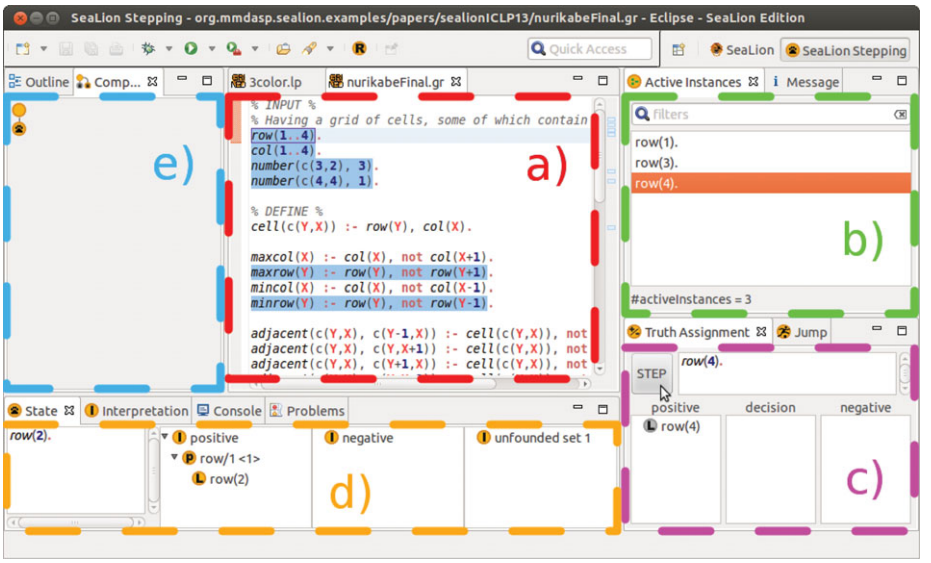

Fig. 3. (Colour online) *SeaLion stepping perspective.*

(visualised in Fig. 2a) he finds out that it has no answer set. This is not intended as for this input there is a solution (shown in Fig. 2b). This is a subcase of (1), where all intended answer sets are missing.

In the case of missing intended answer sets, a reasonable debugging strategy is trying to reach one of these intended answer sets with stepping. It will not be possible to reach the answer set and the reason, i.e., the bug we are looking for, will become apparent as we will see.

In order to start stepping, `SeaLion` needs to know how the program is launched (what files contain the program, what command-line options are specified). Hence, a launch configuration for the program is launched in *debug mode* instead of the *run mode* for computing answer sets. Then, the GUI of Eclipse switches to the *SeaLion stepping perspective* as shown in Figure 3. The stepping perspective displays the source code of the program in a source editor (Fig. 3a) and the current state S of the stepping process in the *state* view (Fig. 3d). State S consists of the set P_S of ground rules that have been chosen in previous steps, the interpretation I_S of atoms that have already been considered true, and the set I_S^- of atoms that have been considered false. Moreover, the state keeps track of unfounded sets (Faber 2005) needed for stepping through answer-set programs with complexity beyond NP, which we do not consider in this paper. Rules with instances that are active under I_S are highlighted in the editor. Upon clicking on such a rule, its active instances are displayed in the *active instances view* (Fig. 3b). After choosing one of the instances, it has to be decided in the *truth assignment view* (Fig. 3c) whether the literals in it are considered positive or negative and subsequently added either to I_S or I_S^-. Naturally, these two sets must stay disjoint throughout the whole course of stepping. As stepping requires that a rule added in some step remains active and satisfied under $I_{S'}$ of the next state S', this assignment is usually fixed for simple rules. In such cases, `SeaLion`

automatically assigns the literals and enables the "STEP" button. Upon clicking the latter, the next state is computed and displayed in the state view. Finally, the *computation view* (Fig. 3e) shows all actions taken during stepping in the form of a tree and allows for navigating between the states resulting from these actions, i.e., one may return to a previous state and start a new branch in the tree from there, while all the old branches remain intact.

Example 3

In our running example, Leo starts stepping through his program by choosing the launch configuration he created for computing the program's answer sets, but he uses the menu under the debug button instead of the launch button (Fig. B4). Leo chooses to step through the first input fact `row(1..4)`. Facts are always active, and for this one the ground instances `row(1).`, `row(2).`, `row(3).`, and `row(4).` are displayed in the active instances view. Leo chooses the second one. After clicking the "STEP" button in the truth assignment view, the rule is added to P_S and its literal is added to I_S which are displayed in the first two columns of the state view. Adding `row(2)` to I_S activates the rule `maxrow(Y) :- row(Y), not row(Y+1).` with its only active instance where `Y` $= 2$. After adding this instance, the state will contain

$$P_S = \{\texttt{row(2).}, \texttt{maxrow(2) :- row(2), not row(3).}\},$$
$$I_S = \{\texttt{row(2)}, \texttt{maxrow(2)}\}, \text{ and}$$
$$I_S^- = \{\texttt{row(3)}\}.$$

As I_S and I_S^- must stay disjoint, it is impossible to add fact `row(3).` to P_S, which will always be an active instance of `row(1..4)`. That means continuing stepping from this state cannot reach any answer set of the program. Detecting when the computation becomes failed will point to the bug later on, but this time Leo added the rule by mistake. He retracts to the last state by double-clicking the previous node in the computation view. Even though Leo did not detect the bug yet, this little detour gave him insight into why `maxrow(X)` is derived only for `X` $= 4$.

Stepping through a lot of simple rules, like the ones in a define part, would be cumbersome and time-consuming. `SeaLion` offers the *jumping feature* for such cases. It allows to "jump" through a number of rules into a state that would be reached after stepping through all these rules individually. Note that decisions for this part are not under user control (technically, they depend on the first answer set that is generated in a `SeaLion` internal answer-set computation). For instance, if guessing rules are chosen for jumping, the outcome of the guess is not predictable for the user. If the user wants to determine some of the guessed values, he or she can step through the corresponding instances of the guessing rules before jumping.

Example 4

Leo would be bored by clicking the "STEP" button for each instance of the rules in the define part. They have the same active instances in any answer set of the program anyway. So he selects all the facts of the input and all the rules in the

define part

```
cell(c(Y,X)) :- row(Y), col(X).
maxcol(X) :- col(X), not col(X+1).
...
adjacent(c(Y,X), c(Y-1,X)) :- cell(c(Y,X)), not minrow(Y).
...
```

and clicks the "Add Rules for Jump" button in the *jump view* (see Fig. B5). This adds the rules to the set of rules selected for jump displayed in the jump view. Upon clicking the "Jump" button in the jump view, he gets to the state where all active instances of these rules are in P_S, the atoms they infer are in I_S, and the negated atoms in these instances are added to I_S^-.

Now Leo wants to add the literals of `white/1` and `black/1` that form the only correct solution (Fig. 2b). So he starts with the choice rule in the generate part, `{white(C):cell(C)}.`. Note that this rule can only be selected as the predicate `cell/1` is already fully evaluated, otherwise `SeaLion` would issue a warning. The rule has a single active instance with a choice atom containing 16 atoms of predicate `white/1`. Leo has to decide which of these atoms are considered true in the truth assignment view. As the choice atom has no bounds, any selection satisfies the rule. He chooses the atoms `white(c(2,2))`, `white(c(2,3))`, `white(c(3,2))`, and `white(c(4,4))` to be true and the remaining atoms false. Using drag and drop (Fig. B6) or the arrow keys, this selection is made within seconds and Leo presses the "STEP" button to continue.

After jumping through the remainder of the generate part, I_S contains exactly those atoms of predicates `white/1` and `black/1` that encode the intended solution. Leo notices that constraint `:- black(C1), black(C2), not black_reach(C1,C2).` that implements condition (iv) is highlighted in the editor, indicating that it has active instances. In order to deactivate it, Leo jumps through the defining rules of the predicate `black_reach/2`, viz.

```
black_reach(C1,C2) :- black(C1), black(C2), adjacent(C1, C2).
black_reach(C1,C3) :- black_reach(C1,C2), black(C3),
                      adjacent(C2,C3).
```

However, the constraint remains active. Leo examines the first active instance of the constraint in the active instances view and sees that `black_reach(c(1,1),c(2,1))` was not derived even though `c(2,1)` should be reachable from `c(1,1)`. In the search for the reason, Leo backtracks the last jump to examine the active instances of the first rule defining `black_reach/2` that should infer the missing atom because the cells are adjacent. It has many active instances but Leo is interested only in the ones that derive the incriminated atom. So, he filters the instances by setting the variable `C1` by typing `C1=c(1,1).` in the "filters" field (Fig. B7). But there is no active instance deriving the atom! Leo notices that the atom `adjacent(c(1,1),c(2,1))` is missing. So, he backtracks before the jump through the rules defining `adjacent/2` and has a close look at the one that should derive that `c(1,1)` is adjacent to `c(2,1)`,

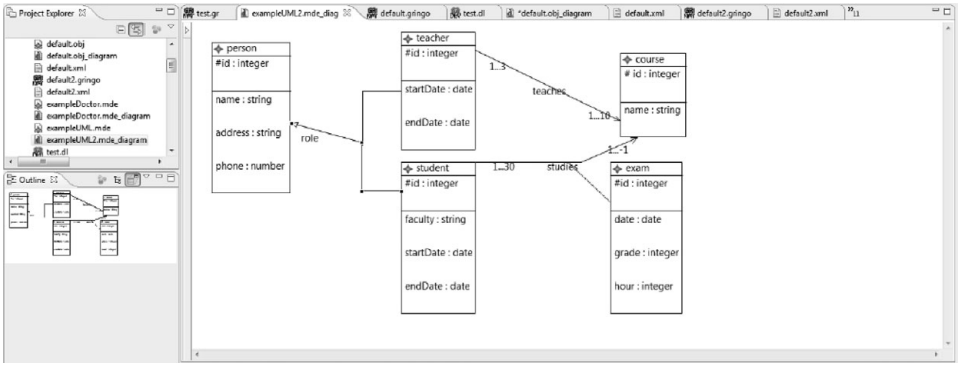

Fig. 4. Modelling in the SeaLion domain diagram editor.

viz.

```
adjacent(c(Y,X), c(Y+1,X)) :- cell(c(Y,X)), not minrow(Y).
```

Leo inspects its active instances and notices that `Y` is never substituted by 1. Consequently, he realises that the boundary atom `not minrow(Y)` is wrong and should be replaced by `not maxrow(Y)`.

3.3 Model-driven engineering framework

`SeaLion`'s MDE plugin allows for guiding the ASP development process by graphical models, starting from modelling the problem domain and ending at the visualisation of problem solutions. That is, we have a graphical editor (cf. Fig. 4) in which the user can start the development by creating a UML class diagram that models the problem domain. In a second step, the model can be translated into an ASP source file that contains LANA annotations documenting the domain. These include descriptions of the predicates and assertions representing constraints expressed in the model such as cardinalities of associations or disjointedness and completeness of generalisations (e.g., that a person is a man or a woman but not both). Moreover, assertions detecting key violations are generated: we allow for defining *key attributes* in our UML diagrams that are not part of the UML standard as in object-oriented languages class instances are typically uniquely identified by an implicit key that represents an address in memory. The translation from the domain model to an ASP predicate schema is similar in spirit to well-known translations from entity-relationship models to database schemas. That is, adding foreign keys to predicates for relationships, mapping every class to a set of predicates, and mapping all attributes to terms. To give flexibility to the user, the mapping is configurable, e.g., the user may choose by how many and by which predicates a class is represented.

After generating the code file, the developer may proceed with completing his or her answer-set program. In the further course of development, the created domain model can be reused for visualising answer sets that use the generated predicate schemas by means of UML object diagrams. That is, based on the UML class diagram and the configuration of the translation, instances of the classes defined in

the model that are encoded in the answer set are detected and displayed (compare Fig. B8). Likewise, their relationships are visualised. Furthermore, the answer set is automatically checked against LANA assertions that were created, and violations of constraints are highlighted in the UML object diagram (Fig. B9). To open the diagram, the user opens a corresponding dialog from the context menu of the answer set that should be visualised directly in the interpretation view. As answer sets can become very large, it is also possible to pre-select an interesting subset of the answer set. In this case, only instances are shown in the diagram whose keys appear in the selected atoms.

3.4 Visualisation and visual editing

Besides the representation in a UML object diagram as presented in the previous section, answer sets can also be visualised without the need of creating a domain model by using the `Kara` plugin (Kloimüllner *et al.* 2011). In the context menu of an interpretation in the interpretation view, one can initiate either a *customised visualisation* or a *generic visualisation.* The latter represents the interpretation as a labelled hypergraph whose nodes are the individuals appearing in the interpretation, and whose edges represent literals in the interpretation, connecting the individuals appearing in a respective literal. An example of a generic visualisation is shown in Figure B10.

The customised visualisation is specified by the user by means of a *visualisation answer-set program* that uses a powerful pre-defined visualisation vocabulary. The resulting visual representation of an interpretation is shown in a graphical editor that also allows for manipulating the visualisation. That is, properties such as colours can be manipulated and graphical elements can be re-positioned, deleted, or even created. Such manipulations are useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG). Second, modifying the visualisation can be used to obtain a modified version of the visualised interpretation by abductive reasoning. In fact, we implemented a feature that allows for abducing an interpretation that would result in the modified visualisation (Kloimüllner *et al.* 2011). Customised visualisations created with `Kara` are given in Figures 2 and B11.

4 Related work

Concerning related approaches, the tool `APE` (Sureshkumar *et al.* 2007), developed at the University of Bath, is also based on Eclipse. It supports the language of the grounder `Lparse` (Syrjänen 2000) and provides syntax highlighting, syntax checking, program outline, and launch configurations. Additionally, `APE` can display the predicate dependency graph of a program.

`ASPIDE`, a recent IDE for `DLV` programs (Febbraro *et al.* 2011), is a standalone tool that builds on previous tools (Calimeri *et al.* 2009; Gebser *et al.* 2009; Febbraro *et al.* 2010). One nice feature of `ASPIDE` is the support of customised code templates. It also supports syntax highlighting, code completion, unit tests (Febbraro *et al.*

2011), and quick fixes. A further feature of ASPIDE is a visual program editor. We do not aim for comprehensive visual source-code editing in SeaLion but consider to use customisable program templates for expressing common programming patterns in future releases of SeaLion. Unfortunately, the profiling component of the IDE (Calimeri *et al.* 2009), that is closely linked with DLV, is not publicly available. Neither APE nor ASPIDE currently support graphical visualisation or visual editing of answer sets as available in SeaLion.

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 only consider the language of either DLV or Lparse, with the exception of iGROM (Koziarkiewicz 2011). 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.

SeaLion is the only IDE offering debugging for programs with variables. However, ASPIDE incorporates the tool Spock (Gebser *et al.* 2009) that is a prototypical debugger for ASP which is limited to ground programs only.

Our model-based engineering plugin is a refined follow-up project of the VIDEAS system (Oetsch *et al.* 2011) that used ER diagrams to model domains of answer-set programs.

Customised visualisation in Kara follows the ideas of the tools ASPVIZ (Cliffe *et al.* 2008) and IDPDraw (Wittocx 2009) that also use ASP for specifying visualisations. Compared to these tools, Kara allows not only for visualisation of an interpretation but also for visually editing the graphical representation such that changes in the visualisation are reflected in the visualised interpretation. Moreover, Kara offers support for generic visualisations, special support for grids, and automatic layout of graph structures. The latter is also the goal of Lonsdaleite, a lightweight script for visualising graph structures encoded in answer sets (Smith 2011).

5 Conclusion and future work

In this paper, we presented SeaLion, an IDE for ASP languages. We discussed general principles that we follow in our implementation and gave an overview of its most important features. SeaLion is an Eclipse plugin and the first comprehensive IDE that supports the languages of both Gringo and DLV, which can currently be considered as the two most prominent implemented ASP languages. Currently, a new standard for solver languages is under development (Calimeri *et al.* 2012) that we want to support in the near future. Other plans for further extensions of SeaLion include a graphical version of the ASPUnit system (De Vos *et al.* 2012) for executing LANA unit tests and further improvements of the stepping plugin like automatic stepping through the stratified part of a program and rules that do not require further user interaction. We see a challenge in studying program transformations that allow to change the representation of a problem in ASP. Such transformations can be used for new refactoring features that support typical changes during program optimisation. A simple example would be the splitting of

predicates. For more advanced methods, also model transformation techniques from model-driven engineering might prove valuable. Our MDE framework could provide a basis for first steps in this direction.

References

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F. and Schaub, T. 2012. ASP-Core-2, input language format. `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`.

Calimeri, F., Leone, N., Ricca, F. and Veltri, P. 2009. A visual tracer for DLV. In *Proceedings of the 2nd International Workshop on Software Engineering for Answer-Set Programming* (*SEA 2009*).

Cliffe, O., De Vos, M., Brain, M. and Padget, J. A. 2008. ASPVIZ: Declarative visualisation and animation using answer set programming. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), 724–728.

De Vos, M., Kisa, D. G., Oetsch, J., Pührer, J. and Tompits, H. 2012. Annotating answer-set programs in Lana. *Theory and Practice of Logic Programming 12,* 4-5, 619–637.

Eclipse Project. 2013. `http://www.eclipse.org/eclipse`.

Faber, W. 2005. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2005*), Lecture Notes in Computer Science, vol. 3662. Springer, 40–52.

Febbraro, O., Leone, N., Reale, K. and Ricca, F. 2011. Unit testing in ASPIDE. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management and the 25th Workshop on Logic Programming* (*INAP 2011/WLP 2011*), 165–176.

Febbraro, O., Reale, K. and Ricca, F. 2010. A visual interface for drawing ASP programs. In *Proceedings of the 25th Italian Conference on Computational Logic* (*CILC 2010*). CEUR Workshop Proceedings, vol. 598.

Febbraro, O., Reale, K. and Ricca, F. 2011. ASPIDE: Integrated development environment for answer set programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), Lecture Notes in Computer Science, vol. 6645. Springer, 317–330.

Fowler, M. 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional.

Frühstück, M., Pührer, J. and Friedrich, G. 2013. Debugging answer-set programs with Ouroboros – extending the SeaLion plugin. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2013*), Springer. To appear.

Gebser, M., Kaufmann, B. and Schaub, T. 2009. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), Lecture Notes in Computer Science, vol. 5753. Springer, 509–514.

Gebser, M., Pührer, J., Schaub, T., Tompits, H. and Woltran, S. 2009. Spock: A debugging support tool for logic programs under the answer-set semantics. In *Proceedings of the 17th International Conference on Applications of Declarative Programming and Knowledge Management and the 21st Workshop on Logic Programming* (*INAP 2007/WLP 2007*), *Revised Selected Papers*, Lecture Notes in Computer Science, vol. 5437. Springer, 247–252.

Gebser, M., Schaub, T. and Thiele, S. 2007. Gringo: A new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), Lecture Notes in Computer Science, vol. 4483. Springer, 266–271.

Gelfond, M. and Leone, N. 2002. Logic programming and knowledge representation - The A-Prolog perspective. *Artificial Intelligence 138,* 1-2, 3–38.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th Logic Programming Symposium*, MIT Press, 1070–1080.

Järvisalo, M., Berre, D. L., Roussel, O. and Simon, L. 2012. The international SAT solver competitions. *AI Magazine 33,* 1, 89–92.

Kloimüllner, C., Oetsch, J., Pührer, J. and Tompits, H. 2011. `Kara` - A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the 25th Workshop on Logic Programming* (*WLP 2011*), 152–164.

Koziarkiewicz, M. 2011. iGROM. `http://igrom.sourceforge.net/`.

Le Berre, D., Roussel, O. and Simon, L. 2009. SAT 2009 competition. `http://www.satcompetition.org/2009/`.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Oetsch, J., Pührer, J., Seidl, M., Tompits, H. and Zwickl, P. 2011. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), Lecture Notes in Computer Science, vol. 6645. Springer, 382–387.

Oetsch, J., Pührer, J., Skočovský, P. and Tompits, H. 2013. Stepping in answer-set programming: Handling disjunctions and aggregates, In preparation.

Oetsch, J., Pührer, J. and Tompits, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming 10,* 4-6, 513–529.

Oetsch, J., Pührer, J. and Tompits, H. 2010. Methods and methodologies for developing answer-set programs—Project description. In *Technical Communications of the 26th International Conference on Logic Programming* (*ICLP 2010*), Leibniz International Proceedings in Informatics (LIPIcs), vol. 7. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

Oetsch, J., Pührer, J. and Tompits, H. 2011. Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), Lecture Notes in Computer Science, vol. 6645. Springer, 134–147.

Oetsch, J., Pührer, J. and Tompits, H. 2011. 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 and the 25th Workshop on Logic Programming* (*INAP 2011/WLP 2011*), 141–151.

Oetsch, J., Pührer, J. and Tompits, H. 2012. Stepwise debugging of description-logic programs. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Lecture Notes in Computer Science, vol. 7265. Springer, 492–508.

Polleres, A., Frühstück, M. and Schenner, G. 2013. Debugging non-ground ASP programs with choice rules, cardinality constraints and weight constraints. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2013*), Springer. To appear.

Schmidt, D. C. 2006. Model-Driven Engineering. *IEEE Computer 39,* 2 (february), 41–47.

Smith, A. 2011. Lonsdaleite. `https://github.com/rndmcnlly/Lonsdaleite`.

SURESHKUMAR, A., VOS, M. D., BRAIN, M. AND FITCH, J. 2007. APE: An AnsProlog* environment. In *Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming* (*SEA 2007*), 71–85.

SYRJÄNEN, T. 2000. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

WITTOCX, J. 2009. IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation`.