The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report

Johannes Oetsch, Jörg Pührer[1(✉)], and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, 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, refactoring functionality, and a visual program outline. Further implemented features are a documentation generator, 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 their *answer sets*, provide the solutions of a problem instance (for an overview about ASP, we refer to a survey article by Gelfond and Leone [1] or to the well-known textbook by Baral [2]). 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

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

H. Tompits et al. (Eds.): INAP/WLP 2011, LNAI 7773, pp. 305–324, 2013.
DOI: 10.1007/978-3-642-41524-1_19, © Springer-Verlag Berlin Heidelberg 2013

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 for DLV programs only. 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.

At present, SeaLion is in a beta version that implements important core functionality and some advanced features. 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. A handy implemented refactoring feature allows for uniform and safe renaming of predicates and terms throughout a program and even across multiple files. 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 neither supported by ASPVIZ nor by IDPDraw. The visualisation functionality of SeaLion is itself represented as an Eclipse plugin, called Kara.[1] In this paper, however, we describe only the basic functionality of Kara; a full description is given in a companion paper [23].

SeaLion integrates the documentation generator ASPDoc for ASP that is based on LANA (Language for ANnotating Answer-set programs), an annotation language for structuring, documenting, and testing answer-set programs [24].

The remainder of the paper is outlined as follows. In the next section, we shortly review the ASP solver languages supported by SeaLion. We discuss the general structure of the IDE, design choices regarding the implementation, as well as how to obtain SeaLion in Sect. 3. Section 4 gives an overview about features that are already functional in SeaLion, whereas Sect. 5 provides an outlook over functionality that is planned to be integrated in the future. In Sect. 6, we discuss other systems related to SeaLion, and we conclude in Sect. 7.

2 Supported ASP Languages

As we focus on supporting ASP developers, we deal with concrete solver languages and refer the reader to the textbook by Baral [2] for a formal introduction to ASP.

SeaLion offers support for the two major ASP solver language families, viz. the input language of the grounding tool Gringo that extends the one of the Lparse grounder and the language of the DLV solver. Both share a common basic Prolog-style rule syntax. In brief, an answer-set program consists of rules of the form

$$a_1 \mid \cdots \mid a_l \;:-\; a_{l+1}, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol "$-$"), and "$|$" is the disjunction symbol (DLV additionally allows for denoting disjunction by the letter "v"). 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 $n = l$, 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.

Despite the common basic rule syntax, the languages of Gringo and DLV differ substantially when it comes to extended features. For one thing, aggregation in Gringo is realised by weight constraint literals that assign weights to literals such that the sum of the weights of true literals must lie between given bounds. For example, consider the weight literal

```
2 [a=1, b=1, c=3] 4,
```

[1] 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.

assigning atoms `a` and `b` weight 1 and atom `c` weight 3. The weight literal is true when the sum of the weights of true atoms is between 2 and 4, i.e., when `a` and `b` are true but `c` is not, or if `c` and at most one of `a` and `b` are true. Aggregates in `DLV`, on the other side, are based on functions over so-called *symbolic sets* that are pairs of (a list of) variables and a conjunction of literals in which these variables appear. For example, the aggregate

```
2 <= #sum{X : a(X)} <= 4
```

is true if the sum of all (integer) constants `c` such that `a(c)` is true is between 2 and 4. Hence, the aggregate is, e.g., true if `a(1)` and `a(3)` but no other atom of predicate `a` is true. As can be seen from the example, `DLV` aggregates require the use of variables but `Gringo` weight constraint literals assign weights to ground literals. Variables in weight constraints are handled using so-called *conditions* that can also be used for ordinary literals in `Gringo` but are not available in `DLV`. For example, during grounding, the literal `redEdge(X,Y):edge(X,Y):red(X):red(Y)` in the body of a rule is replaced by the list of all literals `redEdge(n1,n2)`, where `edge(n1,n2)`, `red(n1)`, and `red(n2)` can be derived.

Further syntactic differences between the languages of `Gringo` and `DLV` are related to finding optimal answer sets. `DLV` uses special rules, called *weak constraints*, for optimisation, while `Gringo` uses minimise and maximise statements. While filtering atoms in the output can be done by hide and show statements in the case of `Gringo`, command-line arguments are needed in the case of `DLV`. For a more detailed description of the solver languages, we refer to the respective user manuals [25,26].

3 Implementation Principles, Architecture, and Availability

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 only 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, 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, optimisation, and filtering support, are also reflected by different classes for `Gringo` and `DLV`, respectively. However, whenever possible, these classes are

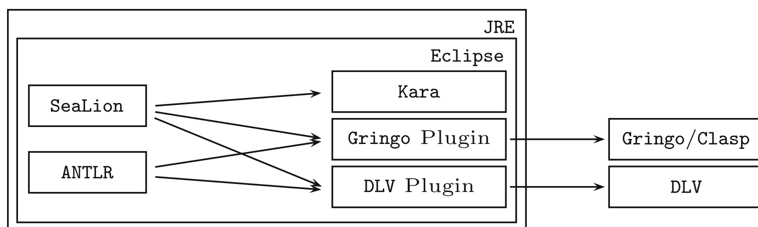

Fig. 1. Technology stack of SeaLion. An arrow indicates that a module is required by another.

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: the main plugin, a plugin that adapts the ANTLR parsing framework [27] to our needs, two solver plugins, one for supporting Gringo/Clasp and one for DLV, and the Kara plugin for answer-set visualisation and visual editing. Figure 1 depicts the technology stack of SeaLion, illustrating the embedding in Eclipse and the Java Runtime Environment (JRE), the aforementioned plugins, as well as the use of answer-set solvers as external applications.

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 refactoring and 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 [28]. 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.

`SeaLion` is free software published under the GNU General Public License version 3. For more information on `SeaLion` and installation instructions we refer to the project web site

http://www.sealion.at.

4 Current Features

In this section, we describe the features that are already operational in `SeaLion`, including technical details on the implementation.

4.1 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. 2. By default, files

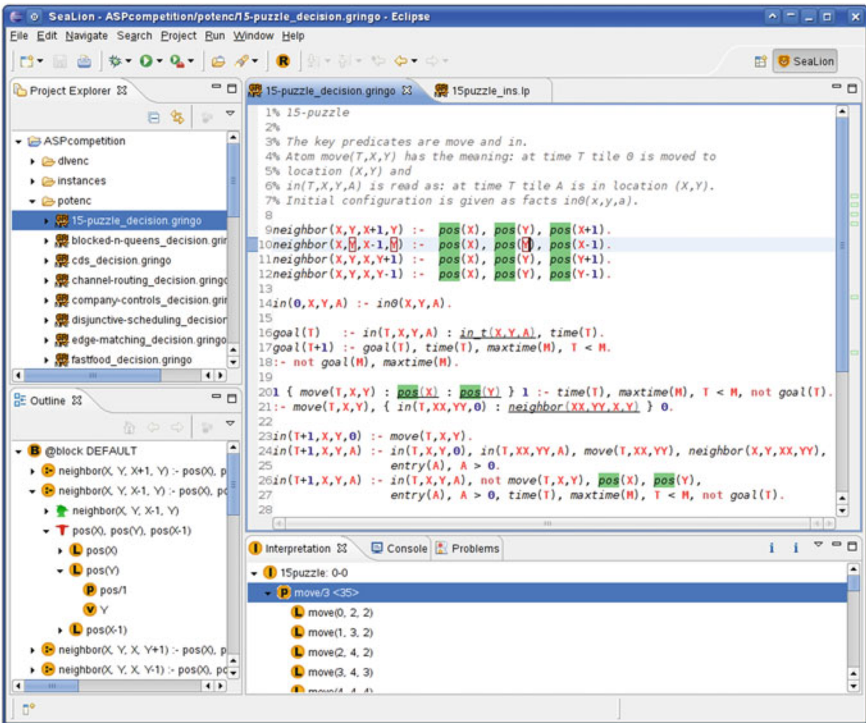

Fig. 2. A screenshot of `SeaLion`'s editor, the program outline, and the interpretation view.

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.

The parsers used are based on the ANTLR framework [27] 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 a user that a constant was used 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 single-line meta commands and "\%*!" and "*\%" for the start and end of block meta commands in the Gringo editor, respectively. Currently, meta commands are used for 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 an ancillary 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 program in a tree-shaped graphical representation. The rules of the programs are represented by nodes of this tree. By expanding the descendant nodes of an individual rule node, one can see its elements, i.e., head, body, literals, predicates, terms, etc.

(cf. Fig. 2). 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 *Eclipse 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 and 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.

A particular application of Eclipse annotations is *problem reporting*. Syntax errors and warnings are displayed in two ways. First, they are marked in the source code 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 a mouse click.

SeaLion offers initial functionality for *refactoring* answer-set programs. Refactoring is the process of improving the source code of a program, e.g., by enhancing its structure, reusability, or readability, without changing its external behaviour. 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. To initiate renaming, the user can either select the targeted program element in the Outline View or place the cursor on it within the editor and use the menu or a keyboard shortcut to open the renaming dialog. On the dialog's first page, the user can specify the new name for the program element and select the files in which renaming should take place. When renaming variables, however, the latter choice is not available because variables are only renamed within a rule and therefore within the same file in which the chosen variable appears. The motivation is that two variables with the same identifier in different rules often have a different meaning. The renaming dialog warns the user if the new name of the program element is already in use anywhere else in the selected programs. As such a renaming still could be intended, it is possible to perform renaming, nevertheless. Once the new 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 Fig. 3.

4.2 Documentation Feature

SeaLion allows for automatically generating source code documentation for answer-set programs, similar as tools like JavaDoc or Doxygen do for other programming languages. For this purpose, the IDE incorporates the ASPDoc documentation generator, a recently developed tool that takes annotated ASP code as input and produces HTML files as output, based on the LANA annotation

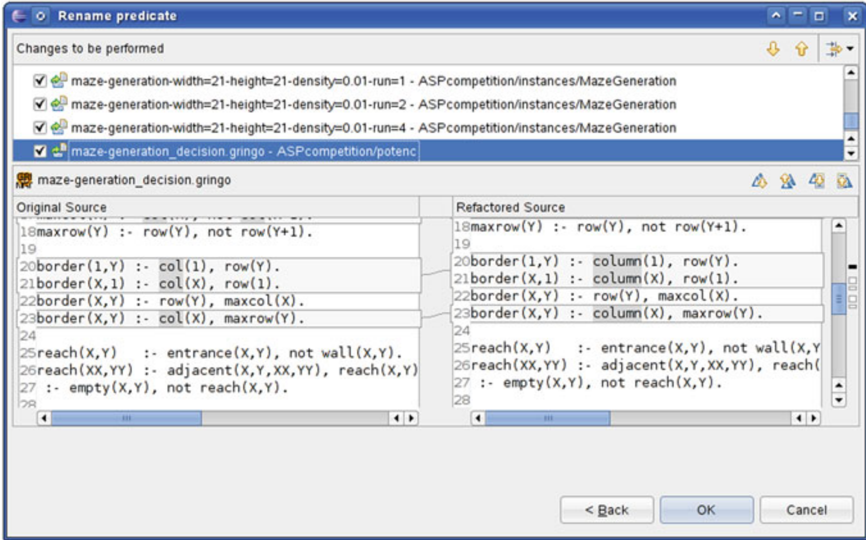

Fig. 3. Reviewing file changes implied by renaming predicate `col/2` to `column/2`.

language [24]. Lana is designed to support the development of answer-set programs even beyond documentation, allowing to group rules into coherent blocks and to specify language signatures, types, pre- and postconditions, as well as unit tests for such blocks. Similar to meta commands in SeaLion, these annotations are invisible to an ASP solver since they have the form of program comments, but they can be interpreted by specialised support tools, e.g., for testing and verification purposes or for eliminating sources of common programmer errors by realising syntax checking or code-completion features. The following example code demonstrates Lana annotations for grouping ASP code into blocks and describing predicates and their arguments using `@atom` and `@term` tags of Lana:

```
%* @block maze {
%*  This is the main block of the maze generation program.
%*  @atom entrance(R,C) gives the position of the maze entrance
%*  @term R is a row index
%*    @with 0 < R, R < 20
%*  @term C is a column index
%*    @with 0 < R, R < 20
%*  ...
    empty(R,C) | wall(R,C) :- row(R),col(C).
    ...
%* }
```

ASP documentation generation can be accessed through Eclipse's export menu. After selecting the ASP programs that should be documented and a target directory, different HTML files are created with `index.html` as the entry point as usual. 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 pre- and postconditions are given. By default, hidden atoms, i.e., atoms never mentioned in a blocks input nor in its output signature, are displayed if the user does not decide otherwise. The documentation also includes HTML versions of the programs' 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.

`SeaLion` can already parse ASP code annotated by the full Lana language. Besides the already implemented documentation functionality, it is planned to integrate further features based on Lana annotations as described in Sect. 5.

4.3 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

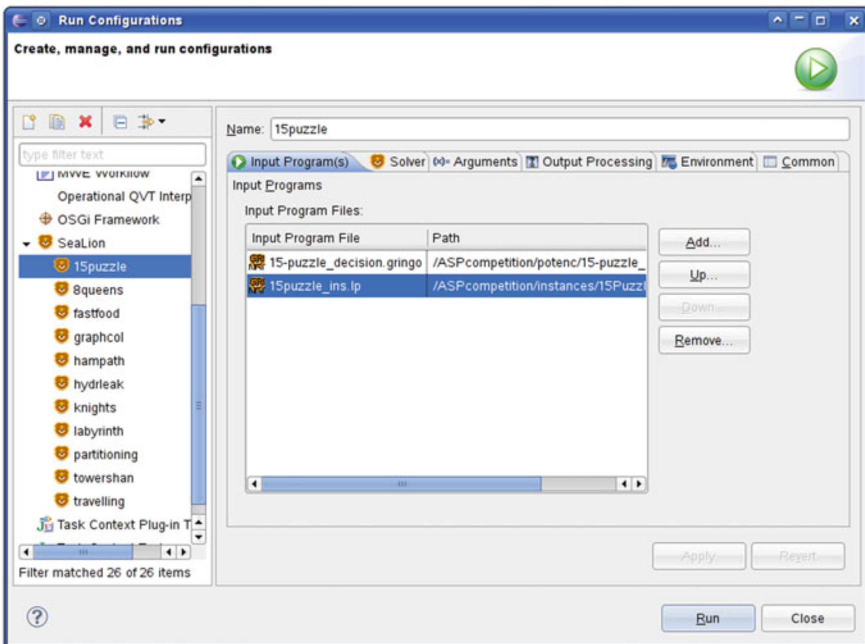

Fig. 4. Selecting two source files for ASP solving in Eclipse's launch configuration dialog.

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 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. Default solvers for different 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, 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 4 shows 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. 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 user can specify the format in which the answer sets are expected from the solver (as there is no standardised form for displaying answer sets). Here, default strategies are preselected, depending on the chosen solver configuration.

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 select 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.

4.4 Interpretation Views

When the user decides to parse answer sets obtained from the solvers, they are stored in SeaLion's *interpretation view* as well as the *interpretation compare view* that is depicted in Fig. 5. In both, interpretations are visualised as expandable trees of depth 3. The root node is the interpretation (marked by an "I") and its children are the predicates (marked by a "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 an "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. While the interpretation view lists interpretations

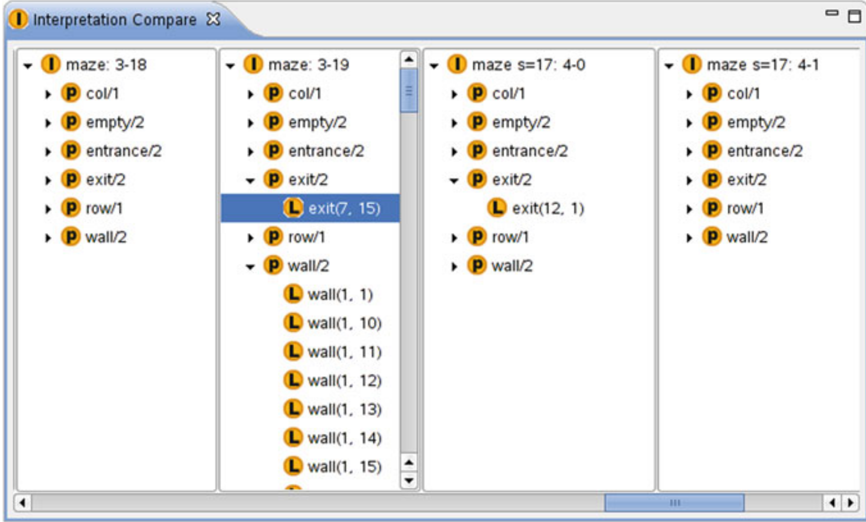

Fig. 5. `SeaLion`'s interpretation compare view.

in rows, the interpretation compare 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.

The two interpretation views are not only meant to provide a good visualisation of results but also serve as a starting point for ASP developing tools that depend on interpretations. One convenient feature is dragging interpretations or individual literals from the interpretation views and dropping them on the source-code editor. When released, these are transformed into facts of the respective ASP language.

4.5 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 choosing 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.

In the generic visualisation, the nodes of the hypergraph are the individuals appearing in the interpretation. Each edge represents a literal 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

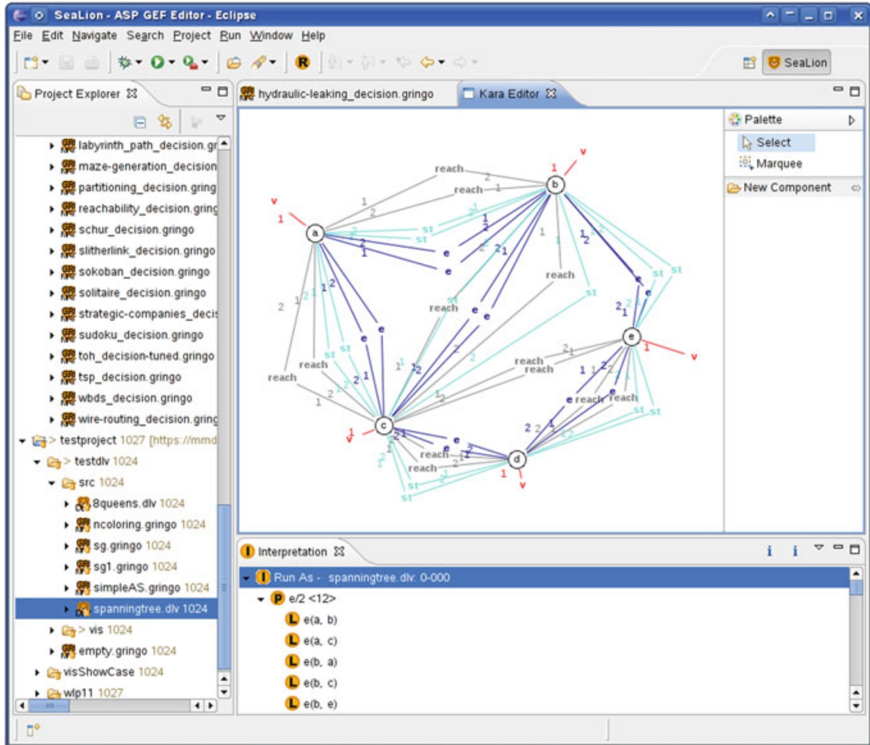

Fig. 6. A screenshot of SeaLion's visual interpretation editor.

tree interpretation is shown in Fig. 6 (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 to be a set of facts) and evaluated using an answer-set solver. One of the resulting answer sets 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. Such manipulations are useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG) by means of our SVG export

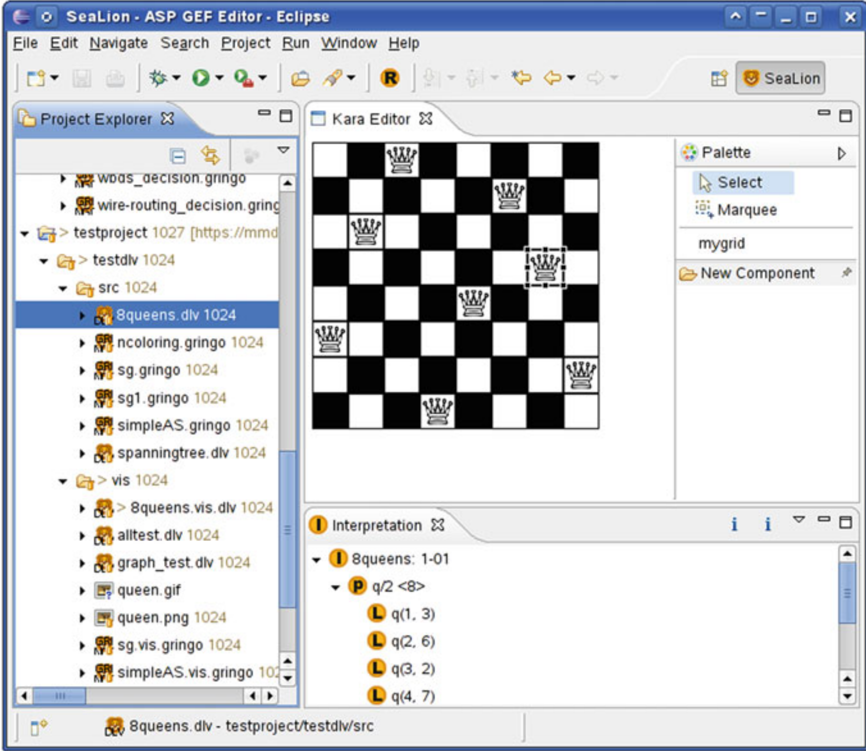

Fig. 7. A customised visualisation of an 8-queens instance.

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 I'_V of the answer set I_V representing the visualisation. We then construct an answer-set program $\lambda(I'_V, \Pi_V)$, 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. 7.

5 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

[29]. 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 between developing programs at the non-ground level and obtaining solutions based on their groundings makes the realisation of practical debugging tools for ASP non-trivial.

As mentioned earlier, our goal is to develop more `SeaLion` features, besides the already implemented documentation generator, exploiting Lana annotations in answer-set programs. Here, one point is that we want to enrich `SeaLion` with support for *typed predicates* which can be specified using Lana. That is, the user can define the domain for a predicate. For instance, consider the predicate `age/2` stating the age of a person. Then, with typing, we can express that for every atom `age(p,a)`, the term `p` represents an element from a set of persons, whereas `a` represents an integer value. Two types of domain specifications will be supported, viz. 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. Moreover, assume we have type specifications stating that for `serves(R,D,P)` and `dishAvailable(D,R)`, `R` is of type `restaurant` and `D` is 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. This way, the programmer would notice that the order of variables in `dishAvailable(R,D)` was mixed up.

In order to avoid problems like in the above example in the first place, auto-completion functionality could be implemented such that variables and constants of correct types are suggested when writing the arguments of a literal in a rule.

The annotation language Lana allows for combining 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 automatically checking pre- and postconditions of answer-set programs

that are defined in Lana. 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 [12,30]. Also, more advanced testing and verification functionality can be realised, like the automatic generation of valid input (with respect to the preconditions) that violates a postcondition.

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.

Besides assertions, it is also planned to offer further testing techniques. In particular, we aim at a unit testing system by integrating the recently developed command-line testing tool `ASPUnit` [24]. The idea is to formulate test cases in the form of ASP programs with Lana annotations that contain information about the expected results under a given reasoning mode when the test-case program is joined with the units under test. Here, units are understood as blocks of ASP rules that are defined using Lana. Multiple test cases can be combined to test suites according to the user's needs. When a test suite is evaluated, `SeaLion` shall give information about what conditions in which test cases failed and, if possible, provide information why.

Other projected features include typical amenities of Eclipse editors like auto-completion, pretty-printing, further means for refactoring, and providing quick-fixes for typical problems in the source code. Also, checks for errors and warnings that are not already done by the parser, e.g., detection of 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) [31], which is a W3C recommendation for exchanging rules in the context of the Semantic Web. Notably, a RIF dialect for ASP, called RIF-CASPD, has been proposed [32].

Further convenience improvements for using external tools in `SeaLion` include 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.

6 Related Work

We next give a short overview of existing IDEs for core ASP languages. To begin with, the tool APE [14], developed at the University of Bath, 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 [33–35]. Some functionality we want to incorporate in SeaLion is already supported by ASPIDE, e.g., code completion, unit tests [36], 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. 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 [16] that provides basic syntax highlighting and syntax checking for the languages of both, Lparse and DLV. 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 [37,38].

Compared to the other visualisation tools, 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 visualisations, special support for grids, and automatic layout of graph structures. The latter is also the goal of Lonsdaleite, a recent tool for visualising graph structures encoded in answer-sets [39]. It is realised as a lightweight Python script that maps the atoms in an answer set to the input format of the Graphviz utilities [40].

7 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 features. SeaLion is an Eclipse plugin and is designed to be 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.

As this is an intermediate report, we also discussed which features we plan to incorporate in future work. The most important step in the advancement of the IDE is the integration of an easy-to-use debugging system that is currently under development. Moreover, we want to implement features for defining types, signatures, pre- and postconditions, and unit tests based on the Lana annotation language into SeaLion. One advantage of using Lana is that, in addition to the graphical tools of the IDE, development support can also be provided by respective command-line tools supporting Lana.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - The A-Prolog perspective. Artif. Intell. **138**(1–2), 3–38 (2002)
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver *clasp*: Progress report. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 509–514. Springer, Heidelberg (2009)
4. SAT 2011 competition. http://www.satcompetition.org
5. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proceedings of ASP 2005. http://CEUR-WS.org (2005)
6. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theor. Pract. Logic Program. **9**(1), 1–56 (2009)
7. Syrjänen, T.: Debugging inconsistent answer-set programs. In: Proceedings of NMR 2006, pp. 77–83. Technische Universität Clausthal (2006)
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
9. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 296–311. Springer, Heidelberg (2009)
10. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: on debugging non-ground answer-set programs. Theor. Pract. Logic Program. **10**(4–5), 513–529 (2010)
11. Niemelä, I., Janhunen, T., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceedings of ECAI 2010, pp. 951–956. IOS Press (2010)
12. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: Random vs. structure-based testing of answer-set programs: An experimental comparison. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 242–247. Springer, Heidelberg (2011)

13. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. **35**, 813–857 (2009)
14. Sureshkumar, A., De Vos, M., Brain, M., Fitch, J.: APE: An AnsProlog* environment. In: Proceedings of SEA 2007, pp. 71–85 (2007)
15. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 317–330. Springer, Heidelberg (2011)
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 ICLP 2010, pp. 154–161. Leibniz-Zentrum für Informatik (2010)
18. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3), 499–562 (2006)
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.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 724–728. Springer, Heidelberg (2008)
22. Wittocx, J.: KRR Software: IDPDraw. https://dtai.cs.kuleuven.be/krr/software/visualisation
23. Kloimüllner, C., Oetsch, J., Pührer, J., Tompits, H.: Kara: A system for visualising and visual editing of interpretations for answer-set programs. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) INAP/WLP 2011. LNCS, vol. 7773, pp. 325–344. Springer, Heidelberg (2013)
24. De Vos, M., Kısa, D.G., Oetsch, J., Pührer, J., Tompits, H.: Annotating answer-set programs in LANA. Theor. Pract. Logic Program. **12**(4–5), 619–637 (2012)
25. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. http://sourceforge.net/projects/potassco/files/potassco_guide
26. Bihlmeyer, R., Faber, W., Ielpa, G., Lio, V., Pfeifer, G.: DLV user manual. http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html
27. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf, Frisco (2007)
28. Ierusalimschy, R.: Programming in Lua, 2nd edn. Lua.Org (2006)
29. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 134–147. Springer, Heidelberg (2011)
30. Oetsch, J., Prischink, M., Pührer, J., Schwengerer, M., Tompits, H.: On the small-scope hypothesis for testing answer-set programs. In: Proceedings of KR 2012, pp. 43–53. AAAI Press (2012)
31. Boley, H., Kifer, M. (eds.): RIF framework for logic dialects. W3C (2010) W3C Recommendation 22 June 2010
32. Kifer, M., Heymans, S.: RIF core answer set programming dialect. http://ruleml.org/rif/RIF-CASPD.html (2009)
33. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proceedings of CILC 2010 (2010)

34. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of SEA 2009 (2009)
35. Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: spock: A debugging support tool for logic programs under the answer-set semantics. In: Seipel, D., Hanus, M., Wolf, A. (eds.) INAP 2007. LNCS, vol. 5437, pp. 247–252. Springer, Heidelberg (2009)
36. Febbraro, O., Leone, N., Reale, K., Ricca, F.: Unit testing in *ASPIDE*. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) INAP/WLP 2011. LNCS, vol. 7773, pp. 345–364. Springer, Heidelberg (2013)
37. ontoprise GmbH: OntoStudio 3.0. http://help.ontoprise.de/ (2010)
38. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based system for enterprise ontologies. J. Logic Comput. **19**(4), 643–670 (2008)
39. Smith, A.: Lonsdaleite. https://github.com/rndmcnlly/Lonsdaleite (2011)
40. AT&T Labs Research and Contributors: Graphviz. http://www.graphviz.org/