

Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs^{*}

Christian Kloimüller¹, Johannes Oetsch², Jörg Pührer², and Hans Tompits²

¹ Forschungsgruppe für Industrielle Software (INSO),
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
christian.kloimueller@inso.tuwien.ac.at

² Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. In answer-set programming (ASP), the solutions of a problem are encoded in dedicated models, called *answer sets*, of a logical theory. These answer sets are computed from the program that represents the theory by means of an ASP solver and returned to the user as sets of ground first-order literals. As this type of representation is often cumbersome for the user to interpret, tools like ASPVIZ and IDPDraw were developed that allow for visualising answer sets. The tool Kara, introduced in this paper, follows these approaches, using ASP itself as a language for defining visualisations of interpretations. Unlike existing tools that position graphic primitives according to static coordinates only, Kara allows for more high-level specifications, supporting graph structures, grids, and relative positioning of graphical elements. Moreover, generalising the functionality of previous tools, Kara provides modifiable visualisations such that interpretations can be manipulated by graphically editing their visualisations. This is realised by resorting to abductive reasoning techniques. Kara is part of SeaLion, a forthcoming integrated development environment (IDE) for ASP.

1 Introduction

Answer-set programming (ASP) [1] is a well-known paradigm for declarative problem solving. Its key idea is that a problem is encoded in terms of a logic program such that dedicated models of it, called *answer sets*, correspond to the solutions of the problem. Answer sets are interpretations, usually represented by sets of ground first-order literals.

A problem often faced when developing answer-set programs is that interpretations returned by an ASP solver are cumbersome to read—in particular, in case of large interpretations which are spread over several lines on the screen or the output file. Hence, a user may have difficulties extracting the information he or she is interested in from the textual representation of an answer set. Related to this issue, there is one even harder practical problem: editing or writing interpretations by hand.

Although the general goal of ASP is to have answer sets computed automatically, we identify different situations during the development of answer-set programs in which it would be helpful to have adequate means to manipulate interpretations. First, in declarative debugging [2], the user has to specify the semantics he or she expects in order for the debugging system to identify the causes for a mismatch with the actual semantics. In previous work [3], a debugging approach has been introduced that takes a program P and an interpretation I that is expected to be an answer set of P and returns reasons why I is not an answer set of P . Manually producing such an intended interpretation ahead of computation is a time-consuming task, however. Another situation in which the creation of an interpretation can be useful is testing post-processing tools. Typically, if answer-set solvers are used within an online application, they are embedded as a module in a larger context. The overall application delegates a problem to the solver by transforming it to a respective answer-set program and the outcome of the solver is then processed further as needed by the application. In order to test post-processing components, which may be written by programmers unaware

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

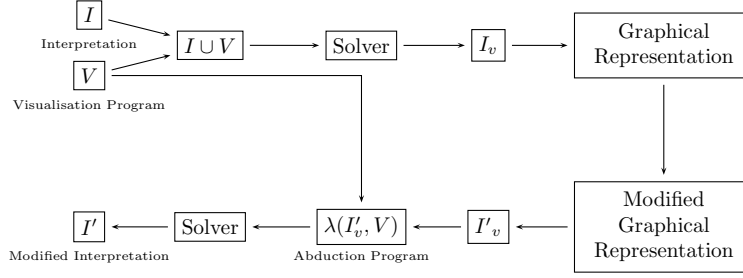


Fig. 1. Overview of the workflow (visualisation and abduction process).

of ASP, it would be beneficial to have means to create mock answer sets as test inputs. Third, the same idea of providing test input applies to modular answer-set programming [4], when a module B that depends on another module A is developed before or separately from A . In order to test B , it can be joined with interpretations mocking answer sets from A .

In this paper, we describe the system `Kara` which allows for both visualising interpretations and editing them by manipulating their visualisations.³ The visualisation functionality of `Kara` has been inspired by the existing tools `ASPVIZ` [5] and `IDPDraw` [6] for visualising answer sets. The key idea is to use ASP itself as a language for specifying how to visualise an interpretation I . To this end, the user takes a dedicated answer-set program V —which we call a *visualisation program*—that specifies how the visualisation of I should look like. That is, V defines how different graphical elements, such as rectangles, polygons, images, graphs, etc., should be arranged and configured to visually represent I .

`Kara` offers a rich visualisation language that allows for defining a superset of the graphical elements available in `ASPVIZ` and `IDPDraw`, e.g., providing support for automatically layouting graph structures, relative and absolute positioning, and support for grids of graphical elements. Moreover, `Kara` also offers a *generic mode* of visualisation, not available in previous tools, that does not require a domain-specific visualisation program, representing an answer set as a hypergraph whose set of nodes corresponds to the individuals occurring in the interpretation.⁴ A general difference to previous tools is that `Kara` does not just produce image files right away but presents the visualisation in form of modifiable graphical elements in a visual editor. The user can manipulate the visualisation in various ways, e.g., change size, position, or other properties of graphical elements, as well as copy, delete, and insert new graphical elements. Notably, the created visualisations can also be used outside our editing framework, as `Kara` offers an SVG export function that allows to save the possibly modified visualisation as a vector graphic. Besides fine-tuning exported SVG files, manipulation of the visualisation of an interpretation I can be done for obtaining a modified version I' of I by means of abductive reasoning [7]. This gives the possibility to visually edit interpretations which is useful for debugging and testing purposes as described above.

In Section 3, we present a number of examples that illustrate the functionality of `Kara` and the ease of coping with a visualised answer set compared to interpreting its textual representation.

`Kara` is designed as a plugin of `SeaLion`, an Eclipse-based integrated development environment (IDE) for ASP [8] that is currently developed as part of a project on programming-support methods for ASP [9].

2 System Overview

We assume familiarity with the basic concepts of answer-set programming (ASP) (for a thorough introduction to the subject, cf. Baral [1]). In brief, an answer-set program consists of rules of the form

$$a_1 \vee \dots \vee a_l :- a_{l+1}, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n,$$

³ The name “`Kara`” derives, with all due respect, from “`Kara Zor-El`”, the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

⁴ A detailed overview of the differences concerning the visualisation capabilities of `Kara` with other tools is given in Section 4.

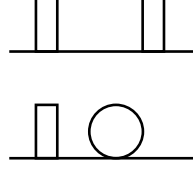


Fig. 2. The visualisation of interpretation I from Example 1.

where $n \geq m \geq l \geq 0$, “not” denotes *default negation*, and all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol, \neg). For a rule r as above, we define the *head* of r as $H(r) = \{a_1, \dots, a_l\}$ and the *positive body* as $B^+(r) = \{a_{l+1}, \dots, a_m\}$. If $n = l = 1$, r is a *fact*, and if $l = 0$, r is a *constraint*. For facts, we will usually omit the symbol “: –”. The *grounding* of a program P relative to its Herbrand universe is defined as usual. An *interpretation* I is a finite and consistent set of ground literals, where consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a . I is an *answer set* of a program P if it is a minimal model of the grounding of the *reduct* of P relative to I (see Baral [1] for details).

The overall workflow of `Kara` is depicted in Fig. 1, illustrating how an interpretation I can be visualised in the upper row and how changing the visualisation can be reflected back to I such that we obtain a modified version I' of I in the lower row. In the following, we call programs that encode problems for which I and I' provide solution candidates *domain programs*.

2.1 Visualisation of Interpretations

As discussed in the introduction, we use ASP itself as a language for specifying how to visualise an interpretation. In doing so, we follow a similar approach as the tools `ASPVIZ` [5] and `IDPDraw` [6]. We next describe this method on an abstract level.

Assume we want to visualise an interpretation I that is defined over a first-order alphabet \mathcal{A} . We join I , interpreted as a set of facts, with a visualisation program V that is defined over $\mathcal{A}' \supset \mathcal{A}$, where \mathcal{A}' may contain auxiliary predicates and function symbols, as well as predicates from a fixed set \mathcal{P}_v of reserved *visualisation predicates* that vary for the three tools.⁵

The rules in V are used to derive different atoms with predicates from \mathcal{P}_v , depending on I , that control the individual graphical elements of the resulting visualisation including their presence or absence, position, and all other properties. An actual visualisation is obtained by post-processing an answer set I_v of $V \cup I$ that is projected to the predicates in \mathcal{P}_v . We refer to I_v as a *visualisation answer set* for I . The process is depicted in the upper row of Fig. 1. An exhaustive list of visualisation predicates available in `Kara` is given in Appendix A.

Example 1. Assume we deal with a domain program whose answer sets correspond to arrangements of items on two shelves. Consider the interpretation $I = \{book(s_1, 1), book(s_1, 3), book(s_2, 1), globe(s_2, 2)\}$ stating that two books are located on shelf s_1 in positions 1 and 3 and that there is another book and a globe on shelf s_2 in positions 1 and 2. The goal is to create a simple graphical representation of this and similar interpretations, depicting the two shelves as two lines, each book as a rectangle, and globes as circles. Consider the following visualisation program:

$$visline(shelf_1, 10, 40, 80, 40, 0). \quad (1)$$

$$visline(shelf_2, 10, 80, 80, 80, 0). \quad (2)$$

$$visrect(f(X, Y), 20, 8) :- book(X, Y). \quad (3)$$

$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- book(s_1, Y). \quad (4)$$

$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- book(s_2, Y). \quad (5)$$

$$visellipse(f(X, Y), 20, 20) :- globe(X, Y). \quad (6)$$

$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- globe(s_1, Y). \quad (7)$$

$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- globe(s_2, Y). \quad (8)$$

⁵ Technically, in `ASPVIZ`, V is not joined with I but with a domain program P such that I is an answer set of P .

Rules (1) and (2) create two lines with the identifiers $shelf_1$ and $shelf_2$, representing the top and bottom shelf. The second to fifth arguments of $visline/6$ represent the origin and the target coordinates of the line.⁶ The last argument of $visline/6$ is a z -coordinate determining which graphical element is visible in case two or more overlap. Rule (3) generates the rectangles representing books, and Rules (4) and (5) determine their position depending on the shelf and the position given in the interpretation. Likewise, Rules (6) to (8) generate and position globes. The resulting visualisation of I is depicted in Fig. 2. \square

Note that the first argument of each visualisation predicate is a unique identifier for the respective graphical element. By making use of function symbols with variables, like $f(X, Y)$ in Rule (3) above, these labels are not limited to constants in the visualisation program but can be generated on the fly, depending on the interpretation to visualise. While some visualisation predicates, like $visline$, $visrect$, and $visellipse$, define graphical elements, others, e.g., $visposition$, are used to change properties of the elements, referring to them by their respective identifiers.

Kara also offers a *generic visualisation* that visualises an arbitrary interpretation without the need for defining a visualisation program. In such a case, the interpretation is represented as a labelled hypergraph. Its nodes are the individuals appearing in the interpretation and the edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of the edge are used for expressing the term position of the individual. To distinguish between different predicates, each edge has an additional label stating the predicate. Edges of the same predicate are of the same colour. A generic visualisation is presented in Example 4.

2.2 Editing of Interpretations

We next describe how we can obtain a modified version I' of an interpretation I corresponding to a manipulation of the visualisation of I . We follow the steps depicted in the lower row of Fig. 1, using abductive reasoning. Recall that abduction is the process of finding hypotheses that explain given observations in the context of a theory. Intuitively, in our case, the theory is the visualisation program, the observation is the modified visualisation of I , and the desired hypothesis is I' .

In Kara, the visualisation of I is created using the Graphical Editing Framework (GEF) [10] of Eclipse. It is displayed in a graphical editor which allows for various kinds of manipulation actions such as moving, resizing, adding or deleting graphical elements, adding or removing edges between them, editing their properties, or change grid values. Each change in the visual editor of Kara is internally reflected by a modification to the underlying visualisation answer set I_v . We denote the resulting visualisation interpretation by I'_v . From that and the visualisation program V , we construct a logic program $\lambda(I'_v, V)$ such that the visualisation of any answer set I' of $\lambda(I'_v, V)$ using V corresponds to the modified one.

The idea is that $\lambda(I'_v, V)$, which we refer to as the *abduction program* for I'_v and V , guesses a set of *abducible atoms*. On top of these atoms, the rules of V are used in $\lambda(I'_v, V)$ to derive a hypothetical visualisation answer set I''_v for I' . Finally, constraints in the abduction program ensure that I''_v coincides with the targeted visualisation interpretation I'_v on a set \mathcal{P}_i of selected predicates from \mathcal{P}_v , which we call *integrity predicates*. Hence, a modified interpretation I' can be obtained by computing an answer set of $\lambda(I'_v, V)$ and projecting it to the guessed atoms. To summarise, the abduction problem underlying the described process can be stated as follows:

- (*) Given the interpretation I'_v , determine an interpretation I' such that I'_v coincides with each answer set of $V \cup I'$ on \mathcal{P}_i .

Clearly, visualisation programs must be written in a way that manipulated visualisation interpretations could indeed be the outcome of the visualisation program for some input. This is not the case for arbitrary visualisation programs, but usually it is easy to write an appropriate visualisation program that allows for abducting interpretations.

The following problems have to be addressed for realising the sketched approach:

- determining the predicates and domains of the abducible atoms, and

⁶ The origin of the coordinate system is at the top-left corner of the illustration window with the x -axis pointing to the right and the y -axis pointing down.

$$\begin{aligned}
\text{dom}(I'_v, V) = & \{ \text{nonRecDom}(t) : -v(\mathbf{t}') \mid r \in V, v/m \in \mathcal{P}_v, v(\mathbf{t}') \in H(r), \\
& a(\mathbf{t}) \in B^+(r), \mathbf{t} = t_1, \dots, t_n, a/n \notin \mathcal{P}_v, \\
& \text{VAR}(t) \neq \emptyset, \text{VAR}(t) \subseteq \text{VAR}(\mathbf{t}') \} \cup \\
& \{ \text{dom}(t) : -v(\mathbf{t}'), \text{nonRecDom}(X_1), \dots, \text{nonRecDom}(X_l) \mid r \in V, \\
& v/m \in \mathcal{P}_v, v(\mathbf{t}') \in H(r), a(\mathbf{t}) \in B^+(r), \mathbf{t} = t_1, \dots, t_n, \\
& a/n \notin \mathcal{P}_v, \text{VAR}(t) \cap \text{VAR}(\mathbf{t}') \neq \emptyset, \\
& \text{VAR}(t) \setminus \text{VAR}(\mathbf{t}') = \{X_1, \dots, X_l\} \} \cup \\
& \{ \text{dom}(X) : -\text{nonRecDom}(X) \}, \\
\text{guess}(V) = & \{ a(X_1, \dots, X_n) : -\text{not } \neg a(X_1, \dots, X_n), \text{dom}(X_1), \dots, \text{dom}(X_n), \\
& \neg a(X_1, \dots, X_n) : -\text{not } a(X_1, \dots, X_n), \text{dom}(X_1), \dots, \text{dom}(X_n) \mid \\
& a/n \notin \mathcal{P}_v, a(t_1, \dots, t_n) \in \bigcup_{r \in V} B(r), \\
& \{ a(t'_1, \dots, t'_n) \mid a(t'_1, \dots, t'_n) \in H(r), r \in V \} = \emptyset \}, \\
\text{check}(I'_v) = & \{ : -\text{not } v(t_1, \dots, t_n), : -v(X_1, \dots, X_n), \text{not } v'(X_1, \dots, X_n), \\
& v'(t_1, \dots, t_n) \mid v(t_1, \dots, t_n) \in I'_v, v/n \in \mathcal{P}_i \},
\end{aligned}$$

Fig. 3. Elements of the abduction program $\lambda(I'_v, V)$.

- choosing the integrity predicates among the visualisation predicates.

For solving these issues, we rely on pragmatic choices that seem useful in practice. We obtain the set \mathcal{P}_a of predicates of the abducible atoms from the visualisation program V . The idea is that every predicate that is relevant to the solution of a problem encoded in an answer set has to occur in the visualisation program if the latter is meant to provide a complete graphical representation of the solution. Moreover, we restrict \mathcal{P}_a to those non-visualisation predicates in V that occur in the body of a rule but not in any head atom in V . The assumption is that atoms defined in V are most likely of auxiliary nature and not contained in a domain program.

An easy approach for generating a domain \mathcal{D}_a of the abducible atoms would be to extract the terms occurring in I'_v . We follow, however, a more fine-grained approach that takes the introduction and deletion of function symbols in the rules in V into account. Assume V contains the rules

$$\begin{aligned}
\text{visrect}(f(\text{Street}, \text{Num}), 9, 10) & :- \text{house}(\text{Street}, \text{Num}) \quad \text{and} \\
\text{visellipse}(\text{sun}, \text{Width}, \text{Height}) & :- \text{property}(\text{sun}, \text{size}(\text{Width}, \text{Height})),
\end{aligned}$$

and I'_v contains $\text{visrect}(f(\text{bakerstreet}, 221b), 9, 10)$ and $\text{visellipse}(\text{sun}, 10, 11)$. Then, when extracting the terms in I'_v , the domain includes $f(\text{bakerstreet}, 221b)$, bakerstreet , $221b$, 9 , 10 , sun , and 11 for the two rules. However, the functor f is solely an auxiliary concept in V and not meant to be part of domain programs. Moreover, the term 9 is introduced in V and is not needed in the domain for I' . Also, the terms 10 and 11 as standalone terms and sun are not needed in I' to derive I'_v . Even worse, the term $\text{size}(10, 11)$, that has to be contained in I' such that I'_v can be a visualisation answer set for I' , is missing in the domain. Hence, we derive \mathcal{D}_a in $\lambda(I'_v, V)$ not only from I'_v but also consider the rules in V . Using our translation that is detailed below, we obtain bakerstreet , $221b$, and $\text{size}(10, 12)$ as domain terms from the rules above.

For the choice of \mathcal{P}_i , i.e., of the predicates on which I'_v and the actual visualisation answer sets of I' need to coincide, we exclude visualisation predicates that require a high preciseness in visual editing by the user in order to match exactly a value that could result from the visualisation program. For example, we do not include predicates determining position and size of graphical elements, since in general it is hard to position and scale an element precisely such that an interpretation I' exists with a matching visualisation. Note that this is not a major restriction, as in general it is easy to write a visualisation program such that aspects that the user wants to be modifiable are represented by graphical elements that can be elegantly modified visually. For example, instead of representing a Sudoku puzzle by labels whose exact position is calculated in the visualisation program, the language of Kara allows for using a logical grid such that the value of each cell can be easily changed in the visual editor.

We next give the details of the abduction program.

Definition 1. Let I'_v be an interpretation with atoms over predicates in \mathcal{P}_v , V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Moreover, let $\text{VAR}(T)$ denote the variables occurring in T , where T is a term or a list of terms. Then, the abduction program with respect to I'_v and V is given by

$$\lambda(I'_v, V) = \text{dom}(I'_v, V) \cup \text{guess}(V) \cup V \cup \text{check}(I'_v),$$

where $\text{dom}(I'_v, V)$, $\text{guess}(V)$, and $\text{check}(I'_v)$ are given in Fig. 3, and $\text{nonRecDom}/1$, $\text{dom}/1$, and v'/n , for all $v/n \in \mathcal{P}_i$, are fresh predicates.

The idea of $\text{dom}(I'_v, V)$ is to consider non-ground terms t contained in the body of a visualisation rule that share variables with a visualisation atom in the head of the rule and to derive instances of these terms when the corresponding visualisation atom is contained in I'_v . In case less variables occur in the visualisation atom than in t , we avoid safety problems by restricting their scope to parts of the derived domain. Here, the distinction between predicates dom and nonRecDom is necessary to prevent infinite groundings of the abduction program. Note that in general it is not guaranteed that the domain we derive contains all necessary elements for abducting an appropriate interpretation I' . For instance, consider the case that the visualisation program contains a rule $\text{visrect}(\text{id}, 5, 5) : - \text{foo}(X)$, and V together with the constraints in $\text{check}(I'_v)$ require that for all terms t of a domain that can be obtained from I'_v and V , $\text{foo}(t)$ must not hold. Then, there is no interpretation that will trigger the rule using this domain, although an interpretation with a further term t' might exist that results in the desired visualisation. Hence, we added an editor to `Kara` that allows for changing and extending the automatically generated domain as well as the set of abducible predicates.

The following result characterises the answer sets of the abduction program.

Theorem 1. Let I'_v be an interpretation with atoms over predicates in \mathcal{P}_v , V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Then, any answer set I''_v of $\lambda(I'_v, V)$ coincides with I'_v on the atoms over predicates from \mathcal{P}_i , and a solution I' of the abduction problem (*) is obtained from I''_v by projection to the predicates in

$$\{a/n \mid a(t_1, \dots, t_n) \in \bigcup_{r \in V} B(r), \{a(t'_1, \dots, t'_n) \mid a(t'_1, \dots, t'_n) \in H(r), r \in V\} = \emptyset\} \setminus \mathcal{P}_v.$$

2.3 Integration in SeaLion

`Kara` is written in Java and integrated in the Eclipse-plugin `SeaLion` [8] for developing answer-set programs. Currently, it can be used with answer-set programs in the languages of `Gringo` and `DLV`. `SeaLion` offers functionality to execute external ASP solvers on answer-set programs. The resulting answer sets can be parsed by the IDE and displayed as expandable tree structures in a dedicated Eclipse view for interpretations. Starting from there, the user can invoke `Kara` by choosing a pop-up menu entry of the interpretation he or she wants to visualise. A run configuration dialog will open that allows for choosing the visualisation program and for setting the solver configuring to be used by `Kara`. Then, the visual editor opens with the generated visualisation. The process for abducting an interpretation that reflects the modifications to the visualisation can be started from the visual editor's pop-up menu. If a respective interpretation exists, one will be added to `SeaLion`'s interpretation view.

The sources of `Kara` and the alpha version of `SeaLion` can be downloaded from

<http://sourceforge.net/projects/mmdasp/>.

An Eclipse update site will be made available as soon as `SeaLion` reaches beta status.

3 Examples

In this section, we provide examples that give an overview of `Kara`'s functionality. We first illustrate the use of logic grids and the visual editing feature.

```

visgrid(maze, MAXR, MAXC, MAXR*20+5, MAXC*20+5):- maxC(MAXC), maxR(MAXR). (9)
visposition(maze, 0, 0, 0). (10)
% A cell with a wall on it.
visrect(wall, 20, 20). (11)
visbackgroundcolor(wall, black). (12)
% An empty cell.
visrect(empty, 20, 20). (13)
visbackgroundcolor(empty, white). (14)
viscolor(empty, white). (15)
% Entrance and exit.
visimage(entrance, "entrance.jpg"). (16)
visscale(entrance, 18, 18). (17)
visimage(exit, "exit.png"). (18)
visscale(exit, 18, 18). (19)
% Filling the cells of the grid.
visfillgrid(maze, empty, R, C):- empty(C, R), not entrance(C, R), not exit(C, R). (20)
visfillgrid(maze, wall, R, C):- wall(C, R), not entrance(C, R), not exit(C, R). (21)
visfillgrid(maze, entrance, R, C):- entrance(C, R). (22)
visfillgrid(maze, exit, R, C):- exit(C, R). (23)
% Vertical and horizontal lines.
visline(v(0), 5, 5, 5, MAXR * 20 + 5, 1):- maxR(MAXR). (24)
visline(v(C), C*20+5, 5, C*20+5, MAXR*20+5, 1):- col(C), maxR(MAXR). (25)
visline(h(0), 5, 5, MAXC * 20 + 5, 5, 1):- maxC(MAXC). (26)
visline(h(R), 5, R * 20 + 5, MAXC * 20 + 5, R * 20 + 5, 1):- row(R), maxC(MAXC). (27)
% Define possible grid values for editing.
vispossiblegridvalues(maze, wall). (28)
vispossiblegridvalues(maze, empty). (29)
vispossiblegridvalues(maze, entrance). (30)
vispossiblegridvalues(maze, exit). (31)

```

Fig. 4. Visualisation program for Example 2.

Example 2. *Maze-generation* is a benchmark problem from the second ASP competition [11]. The task is to generate a two-dimensional grid, where each cell is either a wall or empty, that satisfies certain constraints. There are two dedicated empty cells, being the maze's entrance and its exit, respectively. The following facts represent a sample answer set of a maze generation encoding restricted to interesting predicates.

```

col(1..5). row(1..5). maxC(5). maxR(5). wall(1, 1). empty(1, 2). wall(1, 3).
wall(1, 4). wall(1, 5). wall(2, 1). empty(2, 2). empty(2, 3). empty(2, 4). wall(2, 5).
wall(3, 1). wall(3, 2). wall(3, 3). empty(3, 4). wall(3, 5). wall(4, 1). empty(4, 2).
empty(4, 3). empty(4, 4). wall(4, 5). wall(5, 1). wall(5, 2). wall(5, 3). empty(5, 4).
wall(5, 5). entrance(1, 2). exit(5, 4).

```

Predicates *col/1* and *row/1* define indices for the rows and columns of the maze, while *maxC/1* and *maxR/1* give the maximum column and row number, respectively. The predicates *wall/2*, *empty/2*, *entrance/2*, and *exit/2* determine the positions of walls, empty cells, the entrance, and the exit in the grid, respectively. One may use the visualisation program from Fig. 4 for maze-generation interpretations of this kind.

In Fig. 4, Rule (9) defines a logic grid with identifier *maze*, *MAXR* rows, and *MAXC* columns. The fourth and fifth parameter define the height and width of the grid in pixel. Rule (10) is a fact that defines a fixed position for the maze. The next step is to define the graphical objects to be displayed in the grid. Because these objects are fixed (i.e., they are used more than once), they can be defined as facts. A wall is

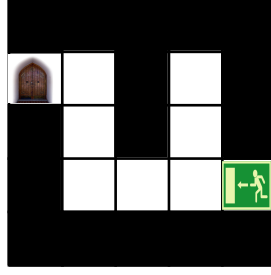


Fig. 5. Visualisation output for the maze-generation program.

represented by a rectangle with black background and foreground colour⁷ (Rules (11) and (12)) whereas an empty cell is rendered as a rectangle with white background and foreground colour (Rules (13) to (15)). The entrance and the exit are represented by two images (Rules (16) to (19)). Then, these graphical elements are assigned to the respective cell of the grid (Rules (20) to (23)). Rules (24) to (27) render vertical and horizontal lines to better distinguish between the different cells. Rules (28) to (31) are not needed for visualisation but define possible values for the grid that we want to be available in the visual editor.

Once the grid is rendered, the user can replace the value of a cell with a value defined using predicate *vispossiblegridvalues*/2 (e.g., replacing an empty cell with a wall). The visualisation of the sample interpretation using this program is given in Fig. 5. Note that the visual representation of the answer set is much easier to cope with than the textual representation of the answer set given in the beginning of the example.

Next, we demonstrate how to use the visual editing feature of *Kara* to obtain a modified interpretation, as shown in Fig. 6. Suppose we want to change the cell (3, 2) from being a wall to an empty cell. The user can select the respective cell and open a pop-up menu that provides an item for changing grid-values. A dialog opens that allows for choosing among the values that have been defined in the visualisation program, using the *vispossiblegridvalues*/2 predicate. When the user has finished editing the visualisation, he or she can start the abduction process for inferring the new interpretation. When an interpretation is successfully derived, it is added to *SeaLion*'s interpretation view. \square

Kara supports absolute and relative positioning of graphical elements. If for any visualisation element the predicate *visposition*/4 is defined, then we have fixed positioning. Otherwise, the element is positioned automatically. Then, by default, the elements are randomly positioned on the graphical editor. However, the user can define the position of an element *relative* to another element. This is done by using the predicates *visleft*/2, *visright*/2, *visabove*/2, *visbelow*/2, and *visinfrontof*/2.

Example 3. The following visualisation program makes use of relative positioning for sorting elements according to their label.

visrect(a, 50, 50). (32)

vislabel(a, laba). (33)

vistext(laba, 3). (34)

vispolygon(b, 0, 20, 1). (35)

vispolygon(b, 25, 0, 2). (36)

vispolygon(b, 50, 20, 3). (37)

vislabel(b, labb). (38)

vistext(labb, 10). (39)

visellipse(c, 30, 30). (40)

vislabel(c, labc). (41)

vistext(labc, 5). (42)

element(X) :- *visrect*(X, -, -). (43)

element(X) :- *vispolygon*(X, -, -, -). (44)

element(X) :- *visellipse*(X, -, -). *element* (45)

⁷ Black foreground colour is default and may not be set explicitly.

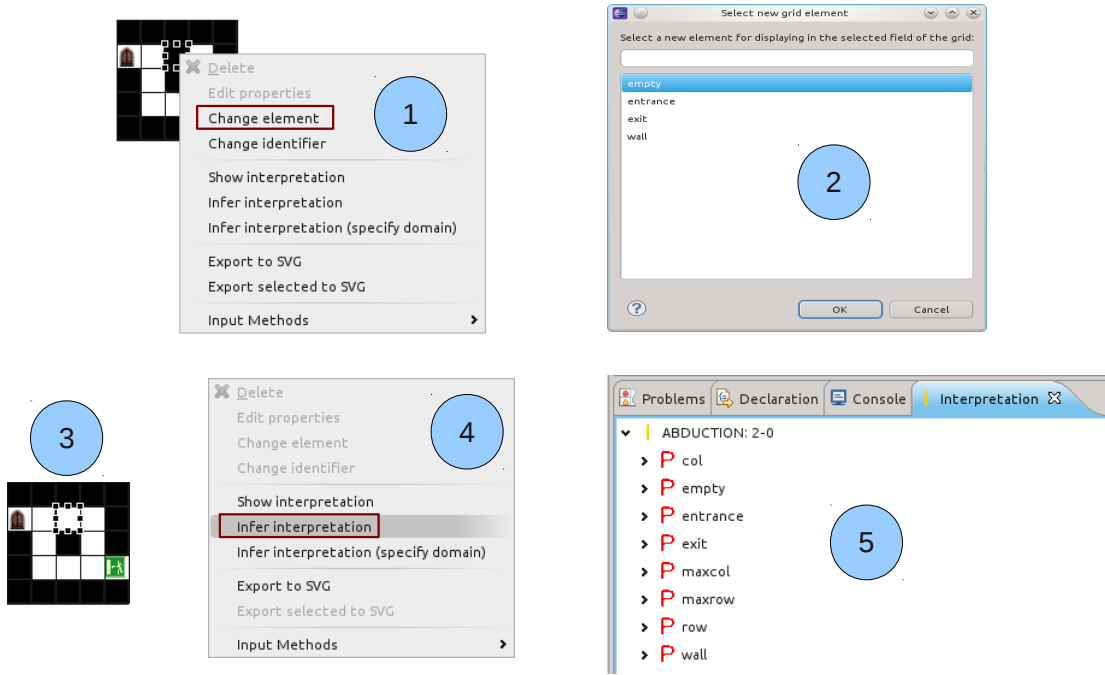


Fig. 6. Abduction steps in the plugin.

$$\begin{aligned}
 visleft(X, Y) :- & \text{element}(X), \text{element}(Y), \text{vislabel}(X, LABX), \\
 & \text{vistext}(LABX, XNUM), \text{vislabel}(Y, LABY), \\
 & \text{vistext}(LABY, YNUM), XNUM < YNUM.
 \end{aligned}
 \tag{46}$$

The program defines three graphical objects, a rectangle, a polygon, and an ellipse. In Rules (32) to (34), the rectangle together with its label 3 is generated. The shape of the polygon (Rules (35) to (37)) is defined by a sequence of points relative to the polygon's own coordinate system using the *vispolygon/4* predicate. The order in which these points are connected with each other is given by the predicate's fourth argument. Rules (38) and (39) generate the label for the polygon and specify its text. Rules (43) to (45) state that every rectangle, polygon, and ellipse is an element. The relative position of the three elements is determined by Rule (46). For two elements E_1 and E_2 , E_1 has to appear to the left of E_2 whenever the label of E_1 is smaller than the one of E_2 . The output of this visualisation program is given in Fig. 7. Note that the visualisation program does not make reference to predicates from an interpretation to visualise, hence the example illustrates that Kara can also be used for creating arbitrary graphics. \square

The last example demonstrates the support for graphs in Kara. Moreover, the generic visualisation feature is illustrated.

Example 4. We want to visualise answer sets of an encoding of a graph-colouring problem. Assume we have the following interpretation that defines nodes and edges of a graph as well as a colour for each node.

$\{node(1), node(2), node(3), node(4), node(5), node(6), edge(1, 2), edge(1, 3), edge(1, 4), edge(2, 4), edge(2, 5), edge(2, 6), edge(3, 1), edge(3, 4), edge(3, 5), edge(4, 1), edge(4, 2), edge(5, 3), edge(5, 4), edge(5, 6), edge(6, 2), edge(6, 3), edge(6, 5), color(1, lightblue), color(2, yellow), color(3, yellow), color(4, red), color(5, lightblue), color(6, red)\}.$

We make use of the following visualisation program:

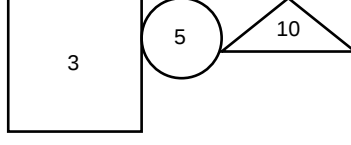


Fig. 7. Output of the visualisation program in Example 3.

```
% Generate a graph.
visgraph(g). (47)
```

```
% Generate the nodes of the graph.
visellipse(X,20,20):- node(X). (48)
```

```
visisnode(X,g):- node(X). (49)
```

```
% Connect the nodes (edges of the input).
visconnect(f(X,Y),X,Y):- edge(X,Y). (50)
```

```
visargetdeco(X, arrow):- visconnect(X,-,-). (51)
```

```
% Generate labels for the nodes.
vislabel(X,l(X)):- node(X). (52)
```

```
visetext(l(X),X):- node(X). (53)
```

```
visfontstyle(l(X), bold):- node(X). (54)
```

```
% Color the node according to the solution.
visbackgroundcolor(X, COLOR):- node(X), color(X, COLOR). (55)
```

In Rule (47), a graph, g , is defined and a circle for every node from the input interpretation is created (Rule (48)). Rule (49) states that each of these circles is logically considered a node of graph g . This has the effect that they will be considered by the algorithm layouting the graph during the creation of the visualisation. The edges of the graph are defined using the *visconnect*/3 predicate (Rule (50)). It can be used to connect arbitrary graphical elements with a line, also if they are not nodes of some graph. As we deal with a directed graph, an arrow is set as target decoration for all the connections (Rule (51)). Labels for the nodes are set in Rules (52) to (54). Finally, Rule (55) sets the colour of the node according to the interpretation. The resulting visualisation is depicted in Fig. 8. Moreover, the generic visualisation of the graph colouring interpretation is given in Fig. 9. \square

4 Related Work

The visualisation feature of *Kara* follows the previous systems *ASPVIZ* [5] and *IDPDraw* [6], which also use ASP for defining how interpretations should be visualised.⁸ Besides the features beyond visualisation, viz. the framework for editing visualisations and the support for multiple solvers, there are also differences between *Kara* and these tools regarding visualisation aspects.

Kara allows to write more high-level specifications for positioning the graphical elements of a visualisation. While *IDPDraw* and *ASPVIZ* require the use of absolute coordinates, *Kara* additionally supports relative positioning and automatic layouting for graph and grid structures. Note that technically, the former is realised using ASP, by guessing positions of the individual elements and adding respective constraints to ensure the correct layout, while the latter is realised by using a standard graph layouting algorithm which is part of the Eclipse framework. In *Kara*, as well as in *IDPDraw*, each graphical element has a unique identifier that can be used, e.g., to link elements or to set their properties (e.g., colour or font style). That way, programs can be written in a clear and elegant way since not all properties of an element have to be specified within a single atom. Here, *Kara* exploits that the latest ASP solvers support function symbols

⁸ *IDPDraw* has been used for visualisation of the benchmark problems of the second and third ASP competition.

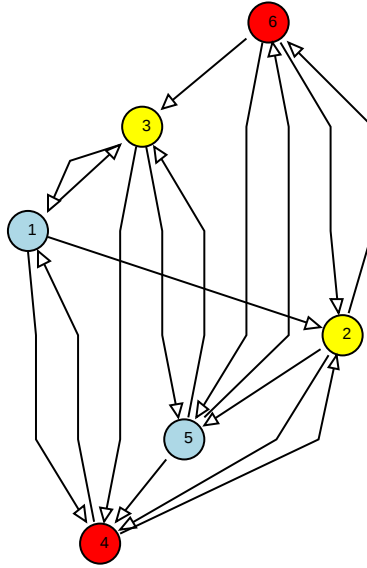


Fig. 8. Visualisation of a coloured graph.

that allow for generating new identifiers from terms of the interpretation to visualise. IDPDraw does not support function symbols. Instead, for having compound identifiers, IDPDraw uses predicates of variable length (e.g., *idp_polygon(id₁, id₂, ...)*). A disadvantage of this approach is that some solvers, like DLV, do not support predicates of variable length. ASPVIZ does not support identifiers for graphical objects.

The support for a *z*-axis to determine which object should be drawn over others is available in Kara and IDPDraw but missing in ASPVIZ. Both Kara and ASPVIZ support the export of visualisations as vector graphics in the SVG format, which is not possible with IDPDraw. A feature that is supported by ASPVIZ and IDPDraw, however, is creating animations which is not possible with Kara so far.

Kara and ASPVIZ are written in Java and depend only on a Java Virtual Machine. IDPDraw, on the other hand, is written in C++ and depends on the qt libraries. Finally, Kara is embedded in an IDE, whereas ASPVIZ and IDPDraw are stand-alone tools.

A related approach from software engineering is the Alloy Analyzer, a tool to support the analysis of declarative software models [12]. Models are formulated in a first-order based specification language. The Alloy Analyzer can find satisfying instances of a model using translations to SAT. Instances of models are first-order structures that can be automatically visualised as graphs, where the nodes correspond to atoms from respective signature declarations in the specification, and the edges correspond to relations between atoms. Since the Alloy approach is based on finding models for declarative specifications, it can be regarded as an instance of ASP in a broader sense. The visualisation of first-order structures in Alloy is closely related to the generic visualisation mode of Kara where no dedicated visualisation program is needed. Alloy supports filtering predicates and arguments away of the graph. We consider to add such a feature in future versions of Kara for getting a clearer generic visualisation.

5 Conclusion

We presented the tool Kara for visualising and visual editing of interpretations in ASP. It supports generic as well as customised visualisations. For the latter, a powerful language for defining a visualisation by means of ASP is provided, supporting, e.g., automated graph layouting, grids of graphical elements, and relative positioning. The editing feature is based on abductive reasoning, inferring a new interpretation as hypothesis to explain a modified visualisation. In future work, we want to add support for defining input and output signatures for programs in SeaLion. Then, the abduction framework of Kara could be easily extended such that instead of deriving an interpretation that corresponds to the modified visualisation, one can derive inputs for a domain program such that one of its answer sets has this visualisation.

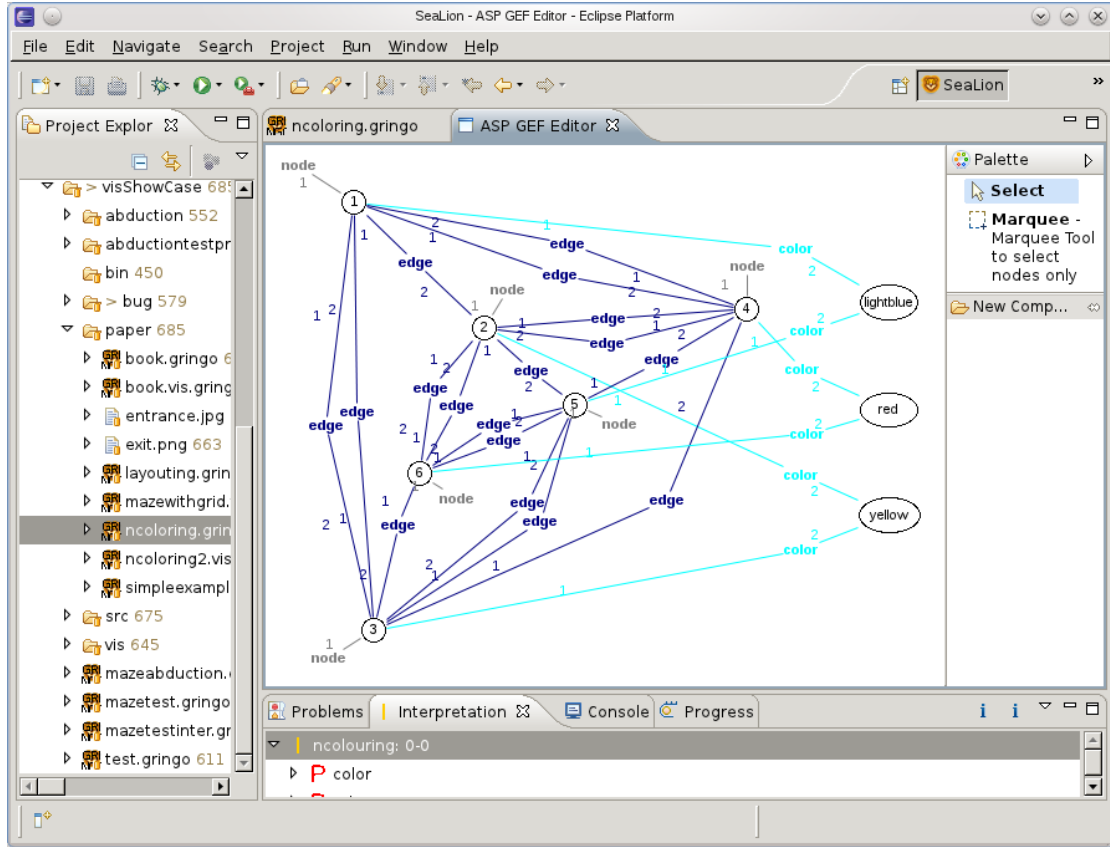


Fig. 9. A screenshot of SeaLion’s visual interpretation editor showing a generic visualisation of the graph colouring interpretation of Example 4 (the layout of the graph has been manually optimised by moving the nodes in the editor).

A Predefined Visualisation Predicates in Kara

Atom	Intended meaning
<i>visellipse</i> (<i>id</i> , <i>height</i> , <i>width</i>)	Defines an ellipse with specified height and width.
<i>visrect</i> (<i>id</i> , <i>height</i> , <i>width</i>)	Defines a rectangle with specified height and width.
<i>vispolygon</i> (<i>id</i> , <i>x</i> , <i>y</i> , <i>ord</i>)	Defines a point of a polygon. The ordering defines in which order the defined points are connected with each other.
<i>visimage</i> (<i>id</i> , <i>path</i>)	Defines an image given in the specified file.
<i>visline</i> (<i>id</i> , <i>x</i> ₁ , <i>y</i> ₁ , <i>x</i> ₂ , <i>y</i> ₂ , <i>z</i>)	Defines a line between the points (<i>x</i> ₁ , <i>y</i> ₁) and (<i>x</i> ₂ , <i>y</i> ₂).
<i>visgrid</i> (<i>id</i> , <i>rows</i> , <i>cols</i> , <i>height</i> , <i>width</i>)	Defines a grid, with the specified number of rows and columns; <i>height</i> and <i>width</i> determine the size of the grid.
<i>visgraph</i> (<i>id</i>)	Defines a graph.
<i>vistext</i> (<i>id</i> , <i>text</i>)	Defines a text element.
<i>vislabel</i> (<i>id</i> _g , <i>id</i> _t)	Sets the text element <i>id</i> _t as a label for graphical element <i>id</i> _g . Labels are supported for the following elements: <i>visellipse</i> /3, <i>visrect</i> /3, <i>vispolygon</i> /4, and <i>visconnect</i> /3.
<i>visisnode</i> (<i>id</i> _n , <i>id</i> _g)	Adds the graphical element <i>id</i> _n as a node to a graph <i>id</i> _g for automatic layouting. The following elements are supported as nodes: <i>visrect</i> /3, <i>visellipse</i> /3, <i>vispolygon</i> /4, <i>visimage</i> /2.
<i>visscale</i> (<i>id</i> , <i>height</i> , <i>weight</i>)	Scales an image to the specified height and width.
<i>visposition</i> (<i>id</i> , <i>x</i> , <i>y</i> , <i>z</i>)	Puts an element <i>id</i> on the fixed position (<i>x</i> , <i>y</i> , <i>z</i>).

<i>visfontfamily</i> (<i>id</i> , <i>ff</i>)	Sets the specified font <i>ff</i> for a text element <i>id</i> .
<i>visfontsize</i> (<i>id</i> , <i>size</i>)	Sets the font size <i>size</i> for a text element <i>id</i> .
<i>visfontstyle</i> (<i>id</i> , <i>style</i>)	Sets the font style for a text element <i>id</i> to bold or italics.
<i>viscolor</i> (<i>id</i> , <i>color</i>)	Sets the foreground colour for the element <i>id</i> .
<i>visbackgroundcolor</i> (<i>id</i> , <i>color</i>)	Sets the background colour for the element <i>id</i> .
<i>visfillgrid</i> (<i>id_g</i> , <i>id_c</i> , <i>row</i> , <i>col</i>)	Puts element <i>id_c</i> in cell (<i>row</i> , <i>col</i>) of the grid <i>id_g</i> .
<i>visconnect</i> (<i>id_c</i> , <i>id_{g₁}</i> , <i>id_{g₂}</i>)	Connects two elements, <i>id_{g₁}</i> and <i>id_{g₂}</i> , by a line such that <i>id_{g₁}</i> is the source and <i>id_{g₂}</i> is the target of the connection.
<i>vissourcedeco</i> (<i>id</i> , <i>deco</i>)	Sets the source decoration for a connection.
<i>vistargetdeco</i> (<i>id</i> , <i>deco</i>)	Sets the target decoration for a connection.
<i>visleft</i> (<i>id_l</i> , <i>id_r</i>)	Ensures that the <i>x</i> -coordinate of <i>id_l</i> is less than that of <i>id_r</i> .
<i>visright</i> (<i>id_r</i> , <i>id_l</i>)	Ensures that the <i>x</i> -coordinate of <i>id_r</i> is greater than that of <i>id_l</i> .
<i>visabove</i> (<i>id_t</i> , <i>id_b</i>)	Ensures that the <i>y</i> -coordinate of <i>id_t</i> is smaller than that of <i>id_b</i> .
<i>visbelow</i> (<i>id_b</i> , <i>id_t</i>)	Ensures that the <i>y</i> -coordinate of <i>id_b</i> is greater than that of <i>id_t</i> .
<i>visinfrontof</i> (<i>id₁</i> , <i>id₂</i>)	Ensures that the <i>z</i> -coordinate of <i>id₁</i> is greater than that of <i>id₂</i> .
<i>vishide</i> (<i>id</i>)	Hides the element <i>id</i> .
<i>visdeletable</i> (<i>id</i>)	Defines that the element <i>id</i> can be deleted in the visual editor.
<i>viscreatable</i> (<i>id</i>)	Defines that the element <i>id</i> can be created in the visual editor.
<i>vischangable</i> (<i>id</i> , <i>prop</i>)	Defines that the property <i>prop</i> can be changed for the element <i>id</i> in the visual editor.
<i>vispossiblegridvalues</i> (<i>id</i> , <i>id_e</i>)	Defines that graphical element <i>id_e</i> is available as possible grid value for a grid <i>id</i> in the visual editor.

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England, UK (2003)
2. Shapiro, E.Y.: Algorithmic Program Debugging. PhD thesis, Yale University, New Haven, CT, USA (May 1982)
3. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. *Theory and Practice of Logic Programming* **10**(4–5) (2010) 513–529
4. Janhunnen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research* **35** (2009) 813–857
5. Cliffe, O., De Vos, M., Brain, M., Padget, J.A.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: *Proceedings of the 24th International Conference on Logic Programming, (ICLP 2008)*. (2008) 724–728
6. Wittoex, J.: IDPDraw, a tool used for visualizing answer sets. <https://dtai.cs.kuleuven.be/krr/software/visualisation> (2009)
7. Peirce, C.S.: Abduction and induction. In: *Philosophical Writings of C.S. Peirce*, Chapter 11. (1955) 150–156
8. Oetsch, J., Pührer, J., Tompits, H.: The SeaLion has landed: An IDE for answer-set programming—Preliminary report. In: *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011)*. (2011)
9. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set programs—Project description. In: Hermenegildo, M., Schaub, T., eds.: *Technical Communications of the 26th International Conference on Logic Programming (ICLP’10)*. Volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
10. The Eclipse Foundation: Eclipse Graphical Editing Framework. <http://www.eclipse.org/gef/>
11. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T., eds.: *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14–18, 2009. Proceedings*. Volume 5753 of *Lecture Notes in Computer Science*, Springer (2009) 637–654
12. Jackson, D.: *Software Abstractions - Logic, Language, and Analysis*. MIT Press (2006)