FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Visualisation and Graphical Editing of Answer Sets: The Kara System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christian Kloimüllner
Matrikelnummer 0628060

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Mag.rer.nat. Dr.techn. Hans Tompits
Mitwirkung: Projektass. Dipl.-Ing. Jörg Pührer

Wien, 14.05.2012

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Christian Kloimüllner
Mariazellerstraße 18, 3100 St. Pölten

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Deutsche Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Visualisierung von Answer Sets für logische Programme. Answer-Set Programmierung ist ein deklaratives Programmierparadigma aus dem Bereich der logikbasierten künstlichen Intelligenz. Obwohl es in der universitären Forschung eine etablierte Methode darstellt, fehlen noch Entwicklungswerkzeuge, die es als Problemlösungstechnik effizienter einsetzbar machen.

Die bereits existierenden Solver berechnen die Answer Sets für ein Answer-Set Programm und geben diese Answer Sets in Form einer Textausgabe am Bildschirm des Programmierers aus. Bei steigender Komplexität der Programme wird meistens auch die Ausgabe der Answer Sets am Bildschirm größer und füllt in der Praxis oftmals mehr als den ganzen Bildschirm aus. Diese Ausgabe ist für den Programmierer selbst nur mehr sehr schwer oder gar nicht mehr überschaubar.

In dieser Arbeit wird auf dieses Problem eingegangen und ein Problemlösungsansatz durch eine bestimmte Visualisierungstechnik erarbeitet, der in dem Tool `Kara` implementiert wird. Das Tool verfolgt die Idee, dass der Benutzer durch das Verfassen eines Visualisierungsprogramms festlegen kann wie ein Answer Set grafisch dargestellt werden soll. Daraus folgt, dass durch die Kombination eines Answer Sets mit dem passenden Visualisierungsprogramm die grafische Darstellung des Answer Sets festgelegt wird, welche einfacher zu interpretieren ist als die direkte Textdarstellung des Solvers.

Dadurch ist es nötig, eine Visualisierungssprache festzulegen mit der der Benutzer die Möglichkeit hat seine erhaltenen Answer Sets grafisch darzustellen. Diese Sprache besteht aus vordefinierten, in dieser Diplomarbeit festgelegten Prädikaten, die für `Kara` eine spezielle Bedeutung haben. Dadurch wird es ermöglicht, dass der Benutzer auf einfache Weise geometrische Formen spezifizieren kann. Das Ergebnis der Kombination eines gegebenen Answer Sets und eines Visualisierungsprogramms ergibt wiederum ein Answer Set, das intern weiterverarbeitet werden kann.

Desweiteren bietet `Kara` auch die Möglichkeit für den Benutzer die Visualisierung grafisch zu bearbeiten. Dadurch können neue Mengen entstehen, die nicht mehr notwendigerweise Answer Sets des gegebenen Answers-Set Programms darstellen. Dies soll einerseits die Fehlersuche für den Benutzer erleichtern und andererseits auch die Möglichkeit bieten, durch das Generieren neuer Answer Sets Testdaten zu erstellen.

Da es es sich nach dem grafischen Bearbeiten der Answer Sets nicht mehr um Answer Sets des ursprünglichen Answer-Set Programms handelt, wird dem Benutzer desweiteren die Möglichkeit geboten, die bearbeitete grafische Darstellung auf ein entsprechendes Answer Set des ursprünglichen Answer-Set Programms rückzurechnen. Dies wird im Zuge der Diplomarbeit auf ein Abduktionsproblem zurückgeführt, wodurch ein Abduktionsprogramm erstellt wird, welches wiederum durch einen Solver ausgeführt werden kann und ein Answer Set in der ursprünglichen Kodierung liefert.

Die Arbeit ist wie folgt gegliedert. In Kapitel 1 wird ein allgemeiner Überblick über Answer-Set Programmierung gegeben sowie die Problematik der Visualisierung von Answer Sets behandelt. Desweiteren werden bereits bestehende Lösungen und Werkzeuge besprochen die auf diese Visualisierungsproblematik eingehen.

Danach wird in Kapitel 2 der formale Hintergrund über Answer-Set Programmierung beschrieben, welcher für die anschließenden Teile der Diplomarbeit notwendig ist. Es werden sowohl syntaktische als auch semantische Aspekte behandelt und Konventionen eingeführt die in dieser Arbeit verwendet werden.

Das darauffolgende Kapitel 3 beschäftigt sich mit der Implementierungssicht des im Zuge dieser Diplomarbeit entwickelten Visualisierungs-Werkzeugs `Kara`. Es werden die einzelnen Module vorgestellt sowie diverse implementierte Algorithmen beschrieben und das System sowie die Technologien als ganzes vorgestellt.

Das Kapitel 4 beschäftigt sich mit der Benutzersicht von `Kara`. Es wird der Ablauf dargestellt um Visualisierungen zu erstellen, diese zu bearbeiten und abschließend auch auf die ausgehende Kodierung rückzurechnen. Im Speziellen wird auch die Sprache genau beschrieben, welche durch die einzelnen Prädikate spezifiziert ist.

Kapitel 5 dient dazu, mehrere Beispiele verschiedener Visualisierungen anzugeben und diese zu beschreiben. Es wird darauf geachtet die meisten definierten Prädikate abzudecken um so deren Verwendung in diversen Visualisierungen praktisch darzustellen.

Kapitel 6 behandelt die bisherigen Visualisierungsansätze in der Literatur. Es werden nicht nur Visualisierungsansätze bezogen auf die Answer-Set Programmierung beschrieben, sondern auch solche für andere Formalismen. Bei eng verwandten Visualisierungsansätzen wird ein genauer Vergleich gegeben und auch ein Beispiel der Visualisierungsprogramme von `Kara` mit dem der anderen Werkzeuge verglichen.

Abschließend wird in Kapitel 7 noch eine Zusammenfassung der Diplomarbeit gegeben und mögliche Erweiterungen für zukünftige Arbeit diskutiert.

Das System `Kara` sowie dessen Verwendung und theoretischer Hintergrund wurden auf dem *25th Workshop on Logic Programming* (*WLP 2011*) in Wien, präsentiert [1]. Diese Arbeit wurde durch Mittel des *Fonds zur Förderung der wissenschaftlichen Forschung* (FWF) unter Projekt Nr. P21698 unterstützt.

Preface

This thesis deals with the visualisation of answer sets, which are the output of answer-set programs. Answer-set programming (ASP) is a fully declarative programming paradigm based on logic programming and non-monotonic reasoning. Although ASP is an acknowledged formalism in logic-based artificial intelligence, development tools for supporting the programmer during coding are missing, which could make it more popular in non-academic settings.

Several highly performant solvers computing answer sets of an answer-set program exist. A characteristic feature of these solvers is that they return these answer-sets as textual output on the screen. When the complexity of the problems or the input instances rises, the output of the solver often gets larger and is probably too large for the user to be analysed or interpreted.

In this thesis, we address this problem by developing an appropriate visualisation system, `Kara`, for answer sets. By writing a visualisation program, the user has the possibility to define the graphical representation of the answer set. Thus, the combination of an answer set with the corresponding visualisation program defines the visualisation of the answer set, which can then be interpreted much easier than the textual output of the solver.

In this thesis, a visualisation language is presented with which the user gets the possibility to specify the graphical representation with ASP itself. The language consists of special predicates dedicated to several shapes and properties of these representations. Using this information of the visualisation program, `Kara` is able to draw the graphical representation of the answer set. The result of combining a given answer set with a visualisation program written by the user outputs in turn an answer set, which, when executed by a solver, can then be processed further by `Kara`.

The tool `Kara` also allows the user to edit the visualisation graphically and thus to easily change properties or even graphical objects encoded in the original answer-set program. While editing, the user may generate new sets of atoms referred to as *interpretations* which are not necessarily answer sets of the original answer-set program. This should ease the debugging process as well and offer the possibility to create new interpretations for test-data generation.

Due to the visualisation language, the answer set of the visualisation is in general not an answer set of the original answer-set program. Thus, `Kara` offers the user the possibility to compute the corresponding interpretation of the original answer-set program from the actual visualisation. The problem of computing the interpretation corresponding to the original program is represented as an abduction problem. This problem is in turn encoded by an abduction program, which—when executed with a solver—outputs the interpretation corresponding to the original program.

The thesis is organised as follows. Chapter 1 gives an overview of answer-set programming in general and outlines the problem of visualising answer sets. Furthermore, existing visualisation tools are presented and described how they addressed the problem of answer-set visualisation.

Then, in Chapter 2, the formal background of answer-set programming is given. The chapter covers syntactic as well as semantic aspects of answer-set programming, and conventions are introduced which are necessary for the subsequent elaboration.

Chapter 3 discusses the implementation view of `Kara`. All modules as well as the implemented algorithms are described in detail.

Chapter 4 describes the user view of the tool. That is, the general workflow how to visualise, edit, as well as computing the corresponding interpretation of the original answer set is given. In particular, the language of `Kara` is described in detail, especially the meaning of the various predefined visualisation predicates of the language.

Afterwards, in Chapter 5, several examples are examined in order to show the practical usage of nearly every important predicate of the visualisation language. Moreover, an example workflow is given how to compute the corresponding interpretation of the original answer set.

Chapter 6 gives an overview of different visualisation approaches for answer sets as well as for other finite first-order language structures. Similar approaches to the one described in this thesis are examined in detail and also example visualisations in the respective visualisation languages are given.

Chapter 7 concludes the thesis and discusses possible extensions for `Kara` as future work.

The tool itself, as well as its application and theoretical background, was presented at the *25th Workshop on Logic Programming* (*WLP 2011*) in Vienna [1]. This thesis was supported by the Austrian Science Fund (FWF) under project P21698.

Acknowledgements

First and foremost, I want to thank my supervisor, Ao.Univ.Prof. Hans Tompits, for his valuable support and guidance during the work on this thesis. In our meetings, we discussed the development and features of the visualisation tool as well as the fine art of writing scientific papers. Thereby, I have learned much about the latter which was tantamount during the writing of this thesis.

Furthermore, I want to thank Jörg Pührer for his support in developing my tool. He designed the core plugin `SeaLion` which is the basis for `Kara`, providing the basic functionality. Moreover, we had many meetings where we discussed the development status of `Kara` as well as how to continue the development of new features. I want to thank him also for his support and help in the theoretical and formal parts of my work.

I furthermore want to thank Johannes Oetsch for his ideas and support concerning theoretical aspects during the development of `Kara`.

Last, but not least, I want to thank my grandparents Franz and Elisabeth for their continuing support during my whole life. Without them it would not be possible for me to study and eventually write this thesis. They supported me in any of my life situations no matter whether they were positive or negative. It is the most valuable thing in life if you know you can count on someone.

Contents

CHAPTER 1

Introduction

Answer-set programming (ASP) is a well-known programming paradigm for declarative problem solving based on logic programming. The programmer of answer-set programs defines the structure of solutions in his or her program but is not responsible for how to get the solution to the problem. As this programming style is based on logic programming, an answer-set program may contain the following parts:

- *facts* defining knowledge which is assumed to be *true*;
- *rules* to derive new knowledge; and
- *constraints* to eliminate invalid solutions.

Answer-set programs are written in a language consisting of predicates from first-order logic, but it is possible to write propositional programs too. Applications of answer-set programming include Semantic-Web reasoning [2, 3], music composition [4], e-tourism [5], and bioinformatics [6, 7].

For obtaining the solutions of a problem defined by an answer-set program, an ASP solver is used. Different solvers are available, which are based on different algorithms. The two most important solvers are `Clasp` [8] and `DLV` [9]. Due to increasing solver efficiency, answer-set programming has gained popularity in academic research.

However, answer-set programming is currently very rarely applied in industrial software projects, arguably due to the absence of tools for developing answer-set programs. Therefore, the academic community for ASP started to address this problem and several papers have been published subsequently as well as approaches for software engineering with ASP were taken into account. Current topics related to software engineering for ASP are

- debugging [10, 11, 12, 13],
- testing [14, 15], and

- modular programming [16, 17, 18].

For debugging, a tool called *spock* [19, 20] has been developed.

The work described in this thesis belongs also to the area of software engineering for ASP. The usual setting when working with an ASP solver is that its output, the *answer sets* of a program, is represented in textual form, containing every literal of the solution, on the screen or in a file. For many problems, these solutions fill the whole screen or are even longer. Therefore, it is arguably a hard task to validate as well as to understand the final answer set(s) of a program. To address this problem, answer-set visualisation tools have been developed. In particular, two such systems have been developed so far, namely `IDPDraw` [21] and `ASPVIZ` [22]. The main idea behind these tools is to visualise interpretations of answer-set programs in a user-friendly way. The programmer declares how the interpretation of the answer-set program is mapped to a suitable graphical representation of the problem, which is done in ASP itself. The output of the tools is a graphical representation of the program according to its domain, which is much easier to validate and to understand than a direct textual output. For instance, `IDPDraw` was used to visualise the output of the *Second ASP Competition* [23].

Moreover, there are sometimes situations where it would be beneficial to create and manipulate answer sets. Such a situation occurs, for example, in *declarative debugging* [24], where the user has to specify expected semantics of a program. There actually was some work [10] on this topic to take a program P and an expected interpretation I of P as input for another program P' whose output gives reasons why the intended interpretation is not an interpretation of P.

Another useful situation to modify answer sets is the *testing of post-processing tools*. In most software engineering applications where ASP is applied, answer-set programs are used as modules in a larger architecture, where problems are relegated to an ASP solver, which returns the solution of a problem. Then, the answer set must be post-processed by the application. However, it would be much better to test these post-processing components with easily created "mock answer sets".

A third use case, also in connection with mock answer sets, would be in *modular answer-set programming* [16]. If a module depends on other modules, which may not be implemented so far, mock answer sets can be used to test the module without waiting until the other modules are realised.

There exist also other visualisation tools, realised for different purposes. To wit, Alloy [25] is a tool for visualising some kind of extended first-order language.[1] It depends on signatures, whereby the user can define types, and supports automatic graph-like visualisation while using the defined signatures as input. Another approach is to visualise the internal structure of satisfiability (SAT) problems for which various graph-layouting algorithms were examined [26].

Our work is based on the approach of `IDPDraw` and `ASPVIZ` to visualise the output of answer-set programs. However, a major difference between our approach and these tools is the possibility in our method to modify the constructed visualisation in an editor and then get the original interpretation from the newly constructed visualisation interpretation. This makes a contribution in debugging answer-set programs as well. Consider the case of a semantically in-

[1] For downloading the tool and a tutorial, see `http://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/index.php`.

correct answer-set program. Then, the interpretation and also the visualisation is erroneous too. The programmer can now modify the visualisation such that it is correct and then infer the original interpretation from the visualisation interpretation and compare the correct interpretation with the semantically incorrect one. This gives evidence for the user which rules are potential reasons for an inaccurate interpretation.

Besides the feature of modifying the visualisation, there are also many enhancements in the visualisation language compared to the previous approaches. Examples are the use of automatic layouting of graphical elements as well as a much richer visualisation language, including the generation of identifiers on the fly, which is possible by exploiting the use of function symbols. A more detailed evaluation of the related tools and our work is given in Chapter 6.

The name of our visualisation and visual editing tool is `Kara`, which is derived from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth. `Kara` is implemented within a larger project which is an `Eclipse` plug-in and is called `SeaLion` [27]. It is intended to be a powerful IDE for answer-set programming supporting all typical features of other IDEs (like for `Java`, `C++`, etc.). To fully integrate `Kara` into this project, the visualisation tool was also implemented as an `Eclipse` plug-in depending on the core features of `SeaLion` for executing and parsing answer-set programs.

`Kara` uses the Graphical Editor Framework (GEF) [28] for visualising and editing the interpretations of answer-set programs. Thereby, the user has the possibility to choose one interpretation he or she likes to visualise. The programmer may define a visualisation program, also written in ASP, which maps the elements of the original interpretation to a visualisation interpretation, which then can be displayed in the graphical editor of `Kara`. The tool offers a rich visualisation language including the possibilities to

- automatically visualise graph structures,
- visualise fixed positioned elements,
- automatically visualise arbitrary components, and
- visualise grids.

Supported graphical elements in this tool are rectangles, polygons, ellipses, images, lines, connections between elements, and text elements. These elements can also have properties like background and foreground colour, line style and width, and many more. A detailed description of the supported graphical elements and their properties will be given later on.

As described above, `Kara` also adds the possibility of modifying the visualised interpretation, which is achieved by using GEF as the underlying framework. Special predicates in the visualisation language of this tool are used to indicate properties of graphical elements, which can be changed in order to modify the visualisation. Furthermore, also the creation and deletion of graphical elements is achieved by special visualisation predicates. After the user has modified the interpretation, it is possible to calculate the original interpretation from the visualisation interpretation by using the related entry in the context menu of the editor.

Besides these visualisation and editing features, `Kara` also supports a convenient export function to create scalable vector graphics (SVG) of the desired visualisations. There is an option

to export the whole content of the editor and one to only export selected graphical elements. This is a nice feature to get portable graphical representations of solutions to the problem.

CHAPTER 2

Background

In this chapter, we describe some formal preliminaries of answer-set programming. Moreover, the problem of visualising answer sets is examined in detail.

2.1 Answer-Set Programming

Answer-set programming [29] is based on logic programming as well as on non-monotonic reasoning and represents a fully declarative way of problem solving. In contrast to conventional programming languages like `Java` or C++, in logic programming it is defined how the result should look like instead of representing how to get the result. `PROLOG` ("programming in logic") is a well known logic-oriented programming language, where the user defines a knowledge base in a fragment of first-order logic. `PROLOG` uses the *closed-world assumption*, meaning that a negated atom $\mathrm{not}\ a$ is true iff a cannot be deduced, which allows for non-monotonic reasoning. A drawback of `PROLOG` is that the body of a rule is processed from left to right, so the order of the atoms in the body influences the results as well as the performance of the evaluation in the proof system. ASP overcomes this drawback by providing full declarativity using the answer-set semantics [30, 31].

The overall approach of the answer-set programming paradigm is depicted in Figure 2.1. The user encodes a problem instance as a logic program, which is then executed by a solver. The output of the solver is given by the answer sets of the problem encoding, which are in a one-to-one correspondence with the solutions of the original problem.

Next, we define syntax and semantics of answer-set programs, and afterwards we give some information on available solver technology.

Syntax of answer-set programs

Answer-set programs use a first-order language for the definition of their knowledge base.

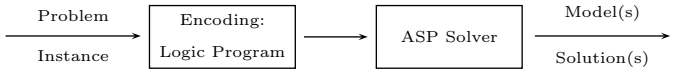

Figure 2.1: ASP as a programming paradigm [29].

Definition 2.1. An *alphabet* $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ consists of a set $\mathcal{P}$ of *predicate symbols*, a set $\mathcal{V}$ of *variables*, and a set $\mathcal{F}$ of *function symbols*. Every predicate and function symbol is associated with an *arity*, defined as the argument count of the predicate or function symbol. For a predicate symbol p and a function symbol f, we write p/n and f/n to indicate that p and f have arity n, respectively. Predicate symbols with arity 0 are called *propositional* and function symbols with arity 0 are called *constants*. We denote the set of all constants by $\mathcal{C}_\mathcal{A}$. □

We use the convention that constant and function symbols always start with lower case letters whereas variables always start with upper case letters. Furthermore, numbers belong to the set of constant symbols. We also use the underline symbol ("_") to refer to an anonymous variable, where we do not need to give an explicit name to a variable.

Definition 2.2. A *string* contains arbitrary text between two quotation marks and is treated exactly the same as a constant symbol. □

Example 2.3. The identifiers $testC$, $test1$, and 123 are all constant symbols, because the first two are starting with a lower case letter and the latter one is a number. Furthermore, the identifier "*foo fighter*" defines a string and thus is also a constant symbol. On the other hand, the identifiers $Test$, X, and $CamelCase$ are variables because every identifier is starting with an upper-case letter. Function symbols with an associated arity can be $f/2, func/2$ or $g/1$. ♢

Definition 2.4. Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ be an alphabet. Then, the set $\mathcal{T}_\mathcal{A}$ of *terms* over $\mathcal{A}$ is defined as follows:

1. Every constant symbol $c \in \mathcal{C}_\mathcal{A}$ and every variable $V \in \mathcal{V}$ is a term.
2. If $t_1, \ldots, t_n$ are terms and $f \in \mathcal{F}$ a function symbol with arity n, then $f(t_1, \ldots, t_n)$ is also a term.
3. The only terms are those constructed by means of Conditions 1 and 2.

□

A predicate with terms as arguments expresses knowledge about the domain. It can define properties, relationship between constant symbols in the domain, etc. We call such an expression an *atom*.

Definition 2.5. Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ be an alphabet. Then, an *atom* a over $\mathcal{A}$ is an expression of form $p(t_1, t_2, \ldots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms over $\mathcal{A}$. Moreover, a *classical literal* is an atom or an atom preceded by the *strong negation operator* $\neg$, and a *default literal*, or simply *literal*, is a classical literal or a classical literal preceded by the *default negation* not. □

An atom $p(t_1, t_2, \ldots, t_n)$ or term $f(t_1, t_2, \ldots, t_n)$ is *ground* iff it contains no variable, otherwise the atom or term is *non-ground*. Analogously, a literal $\text{not } a$ or $\neg a$ is *ground* iff the atom a is *ground*, otherwise the literal is *non-ground*.

Example 2.6. Consider an alphabet $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ with $\mathcal{C}_\mathcal{A} = \{joe, mary\}$, $X, Y \in \mathcal{V}$, $\mathcal{F} = \{father/1\}$, and $\mathcal{P} = \{childOf/2\}$. Then:

- $father(mary)$ and $father(father(mary))$ are ground terms;
- $father(X)$ and $father(father(Y))$ are non-ground terms;
- $childOf(joe, mary)$ is a ground atom;
- $childOf(X, Y)$, $childOf(joe, X)$, and $childOf(mary, Y)$ are non-ground atoms;
- $\text{not } childOf(joe, mary)$ and $\neg childOf(joe, mary)$ are ground literals; and
- $\text{not } childOf(X, Y)$, $\neg childOf(joe, X)$, and $childOf(mary, Y)$ are non-ground literals.

$\diamondsuit$

Definition 2.7. A (*disjunctive*) *rule* is a pair $\langle H, B \rangle$, where H is a set of classical literals and B is a set of default literals such that $H \cup B \neq \emptyset$. A rule $r = \langle H, B \rangle$ is usually written as

$$h_1 \vee h_2 \vee \cdots \vee h_n :- b_1, b_2, \ldots b_m, \text{not } c_1, \text{not } c_2, \ldots, \text{not } c_k, \tag{2.1}$$

for $H = \{h_1, \ldots, h_n\}$ and $B = \{b_1, b_2, \ldots b_m, \text{not } c_1, \text{not } c_2, \ldots, \text{not } c_k\}$.

$\square$

We use the following notation to refer to the separate parts of a rule r of form (2.1):

- $\mathrm{H}(r) = \{h_1, \ldots, h_n\}$, called the *head* of r;
- $\mathrm{B}^+(r) = \{b_1, b_2, \ldots b_m\}$, called the *positive body* of r;
- $\mathrm{B}^-(r) = \{\text{not } c_1, \text{not } c_2, \ldots, \text{not } c_k\}$, called the *negative body* of r; and
- $\mathrm{B}(r) = \mathrm{B}^+(r) \cup \mathrm{B}^-(r)$, called the *body* of r.

Example 2.8. Assume an alphabet $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ with $\mathcal{C}_\mathcal{A} = \{joe, mary, kate\}$, $X, Y, Z \in \mathcal{V}$, $\mathcal{P} = \{childOf/2, grandchildOf/2\}$, and $\mathcal{F} = \{father/1\}$.

Given the rule

$$r = grandchildOf(X, Z) :- childOf(X, Y), childOf(Y, Z),$$

then $\mathrm{H}(r) = \{grandchildOf(X, Z)\}$, $\mathrm{B}(r) = \mathrm{B}^+(r) = \{childOf(X, Y), childOf(Y, Z)\}$, and $\mathrm{B}^-(r) = \emptyset$.

$\diamondsuit$

Next, we define some properties of rules.

Definition 2.9. A rule is a *constraint* iff $\mathrm{H}(r) = \emptyset$, and a *fact* iff $\mathrm{B}(r) = \emptyset$. Furthermore, r is *ground* iff every literal of r is ground, and r is called *propositional* iff every predicate of r has arity 0. □

For facts, we often omit the symbol ":− ", or use a period "." instead (as is supported by the syntax of standard ASP solvers).

Moreover, we introduce the term *safe rule*. It is used to restrict the possible values for variables occurring in the head of a rule such that no general statements about all objects can be made.

Definition 2.10. A rule r is *safe* iff every variable occurring in $\mathrm{H}(r) \cup \mathrm{B}^-(r)$ also occurs in $\mathrm{B}^+(r)$. □

Example 2.11. Consider the following rules:

$$winner(Z) :- \ wonAgainst(X, Y), \tag{2.2}$$
$$canFly(X) :- \ bird(X), \text{not } penguin(X), \tag{2.3}$$
$$:- \ ball(X), \text{not } round(X). \tag{2.4}$$

Rule (2.2) is unsafe because variable Z does not occur in any of its positive body atoms whereas Rule (2.3) is safe because variable X, used in the head as well as in a negative body atom, also occurs in the positive body atom $bird(X)$. Finally, Rule (2.4) is also safe because variable X, used in the negative body atom $round(X)$, is also used in the positive body atom $ball(X)$. ♢

After introducing the safety property of rules we can define *logic programs* thus:

Definition 2.12. A (*disjunctive*) *logic program* is a finite set of safe rules. □

Furthermore, we apply the conditions of rules also on logic programs:

Definition 2.13. A logic program is *ground* iff every rule of this logic program is ground. Likewise, a logic program is *propositional* iff every rule of it is propositional. □

In general, a logic program consists of facts, rules, and constraints. Thus, the variables, domain, function symbols, as well as predicate symbols are given implicitly by their occurrence in the program, which yields the definition of the *Herbrand base* (*HB*) as well as of the *Herbrand universe* (*HU*) of a program.

Definition 2.14. The *Herbrand universe* of a program Π, $HU(\Pi)$, is the set of all terms which can be constructed from the function symbols and the constants occurring in Π. Moreover, the *Herbrand base* of Π, $HB(\Pi)$, is the set of all ground atoms which can be built by using the predicate symbols of Π and the terms in $HU(\Pi)$. □

We use the concept of *grounding* for the process of transforming non-ground programs into ground ones.

Definition 2.15. The *grounding* of a rule r, denoted by $ground(r)$, is the process of uniformly substituting every variable occurring in r by a term of $HU(\Pi)$. The grounding of a logic program Π, $ground(\Pi)$, is obtained by grounding every rule of Π. □

Example 2.16. Consider the following program Π:

$$\begin{aligned}\Pi = \{\ & grandchildOf(X,Z) \ :- childOf(X,Y), childOf(Y,Z),\\ & childOf(mary, john) :- ,\\ & childOf(john, kate) \ :- \}.\end{aligned}$$

Then, an excerpt of $ground(\Pi)$ contains the following rules:

$$\begin{aligned}grandChildOf(mary, john) &:- childOf(mary, kate), childOf(kate, john),\\ grandChildOf(mary, kate) &:- childOf(mary, john), childOf(john, kate),\\ grandChildOf(john, kate) &:- childOf(john, mary), childOf(mary, kate),\\ grandChildOf(john, mary) &:- childOf(john, kate), childOf(kate, mary),\\ grandChildOf(kate, mary) &:- childOf(kate, john), childOf(john, mary),\\ grandChildOf(kate, john) &:- childOf(kate, mary), childOf(mary, john),\\ childOf(mary, john) &:- ,\\ childOf(john, kate) &:- .\end{aligned}$$

♢

Semantics of answer-set programs

After discussing the syntax of answer-set programming, we now examine its semantics. First, we define the notion of an *interpretation*.

Definition 2.17. A set of ground classical literals over $\mathcal{A}$ is *consistent* iff it does not contain both an atom $p(X)$ and its strong negation $\neg p(X)$. □

Example 2.18. The set $I_1 = \{p(1,2,3), \neg p(2,2,2), p(5,2,1)\}$ of ground classical literals is consistent, whereas the set $I_2 = \{p(1,2,2), p(5,2,1), \neg p(1,2,2)\}$ is *not* consistent. ♢

Definition 2.19. Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ be an alphabet. Then, an *interpretation* (*over* $\mathcal{A}$) is a consistent set of ground classical literals over $\mathcal{A}$. □

Furthermore, we need to define the truth value of ground literals:

Definition 2.20. Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ be an alphabet, I an interpretation over $\mathcal{A}$, and l a ground literal over $\mathcal{A}$. Then, $I \models l$ is defined as follows:

- if $l = a$, for an atom a, then $I \models l$ iff $a \in I$;
- if $l = \neg a$, for an atom a, then $I \models l$ iff $\neg a \in I$; and
- if $l = \text{not } q$, for a classical literal q, then $I \models l$ iff $q \notin I$.

Moreover, if $I \models l$, then l is said to be *true under* I, otherwise l is *false under* I. □

Next, we can define the semantics and the applicability of rules in answer-set programs.

Definition 2.21. Let I be an interpretation. A rule r is *applicable under* I if $I \models \mathrm{B}(r)$, otherwise r is *blocked under* I.

A rule r is *satisfied by* I, or r is *true under* I, symbolically $I \models r$, if some $h_i \in \mathrm{H}(r)$ belongs to I or the rule is blocked under I. Otherwise, the rule is *unsatisfied by* I, or *false under* I. □

Definition 2.22. An interpretation I satisfies a program Π, symbolically $I \models \Pi$, iff I satisfies every rule in Π. If $I \models \Pi$, then I is a *model* of Π. □

There can be models of a program Π which contain atoms not necessarily to be true in order to be a model of Π. Thus, we define the notion of an intended model in form of the *minimal-model semantics*.

Definition 2.23. A model I of a program Π is *minimal* if there is no $J \subset I$ which is also a model of Π. □

Example 2.24. Given the program Π, containing the rules

$$fatherOf(george, william) :- \ , \tag{2.5}$$
$$fatherOf(william, michael) :- \ , \tag{2.6}$$
$$grandfatherOf(X, Z) :- \ fatherOf(X, Y), fatherOf(Y, Z), \tag{2.7}$$

and the interpretations

$$\begin{aligned} I_1 = \{&fatherOf(george, william), fatherOf(william, michael),\\ &grandfatherOf(george, michael)\} \text{ and}\\ I_2 = \{&fatherOf(george, william), fatherOf(william, michael),\\ &fatherOf(george, michael), grandfatherOf(george, michael)\}, \end{aligned}$$

I_1 is minimal whereas I_2 is not minimal because $fatherOf(george, michael)$ is not necessarily true. Thus, I_1 is the only least model in this example. ♢

After defining minimal models, we are now in the position to define answer sets [30, 31].

Definition 2.25. Let Π be a ground program and I an interpretation. Then, the *reduct,* Π^I*, of* Π *with respect to* I is given by

$$\Pi^I = \{\mathrm{H}(r) :- \mathrm{B}^+(r) | r \in \Pi, I \models \mathrm{B}(r)\}.$$

□

The purpose of building a reduct of a program is to remove default negation based on a candidate interpretation I.

Definition 2.26. An interpretation I is an *answer set* of a program Π iff I is a minimal model of $ground(\Pi)^I$. We write $AS(\Pi)$ for denoting the set of all answer sets of Π. □

In general, answer-set programs with disjunction and default negation can have no or multiple answer sets.

Example 2.27. Consider the logic program Π_1, containing the following rules:

$$vehicle(mercedes) :- \ , \tag{2.8}$$
$$car(X) \vee bike(X) :- \ vehicle(X). \tag{2.9}$$

Π_1 has exactly two answer sets, namely:

$$AS(\Pi_1) = \{\{vehicle(mercedes), car(mercedes)\}, \{vehicle(mercedes), bike(mercedes)\}\}.$$

Consider program Π_2, comprising the following rules:

$$penguin(dora) :- \ , \tag{2.10}$$
$$\mathit{flying}(X) :- \ bird(X), \tag{2.11}$$
$$bird(X) :- \ penguin(X), \tag{2.12}$$
$$\neg \mathit{flying}(X) :- \ penguin(X). \tag{2.13}$$

Π_2 has no answer set at all. The atom $\mathit{flying}(dora)$ as well as the strongly negated atom $\neg \mathit{flying}(dora)$ are conflicting and thus there is no consistent interpretation which could be an answer set of Π_2.

Lastly, consider the program Π_3, consisting of the single rule

$$legs(X, 4) :- \ goat(X). \tag{2.14}$$

The smallest set of atoms satisfying Π_3 is $\emptyset$ and due to the minimality criterion of answer sets it is also be the only answer set, i.e., $AS(\Pi_3) = \{\emptyset\}$. $\diamondsuit$

By a *positive program* we understand a program without disjunction and negation. The following property is obvious:

Proposition 2.28. *Every positive program has at most one answer set.*

2.2 Solvers

If the programmer has defined an answer-set program for a specific problem and he or she wants to compute the solutions, i.e., the answer sets, of the program, a solver must be used. There are different solvers available, which use different algorithms to compute the answer sets of a program. This is interesting, because it is possible that the various solvers have a different performance on specific problems.

Generally, the syntax of different solvers is mostly the same, but some extensions of the basic language of answer-set programs differ, like for disjunction and aggregate functions. At the moment, the two most important solvers are `Clasp` [8] and `DLV` [9].

There are not only syntactic differences between the solvers but also semantic differences for aggregate functions and furthermore the solvers support different features. `Clasp`, for instance,

does not support disjunction in logic programs whereas DLV does. However, there exists a special version of Clasp, called ClaspD, which does support disjunction. Moreover, the concept of *weak constraints* is also implemented differently. Weak constraints have an associated weight and are used as input to the solver to optimise the produced solutions.

DLV provides the grounder and the solver in one component, whereas Clasp is only a solver and provides no grounding. For grounding the instances as input to Clasp, the tool Gringo is used.

An example program, which can be parsed by both Gringo and DLV is the following:

$$mother(andrea, peter).$$
$$mother(maria, andrea).$$
$$grandmother(GM, C) :- mother(GM, M), mother(M, C).$$

A sample execution of DLV with this answer-set program, stored in file test.dlv, yields the following output:

```
user@host:~$ dlv test.dlv
DLV [build BEN/Oct 14 2010   gcc 4.4.3]

{mother(andrea,peter), mother(maria,andrea),
 grandmother(maria,peter)}
```

Most solvers support the use of disjunctions as well as function symbols in the program encoding. Furthermore, they have built-in arithmetic as well as comparison functions. Another convenient feature is that solvers allow the use of intervals for specifying facts. For instance, the following fact defines that the constants 1 to 100 are numbers:

$$number(1..100).$$

They also support aggregate functions like *count* and *sum*, and in Gringo optimisation functions are supported too, which are syntactically similar to aggregate functions and are called *minimize* and *maximize*. Of course, single and multiple line comments are supported as well. It is also important to mention that the solvers support multiple command-line parameters which allow to set options to the solver on startup. An example is to restrict the set of natural numbers, which can also be set with an meta-parameter in the answer-set program. Gringo also supports an integrated scripting language, lua, which can be called with an "@" prefix in front of the function call, as, e.g., in the rule

$$colour(X, @c(Y)) :- assign(X, Y).$$

The use of different solvers is also supported in our approach and it is possible to use both Clasp and DLV for solving and visualising answer-set programs in Kara.

CHAPTER 3

The Kara System

This chapter gives an overview of the visualisation component, describing the integration in SeaLion, as well as the implementation itself. Moreover, the overall architecture is described and the process for inferring the original interpretation from the visualisation interpretation is given in detail (in theoretical as well as in practical terms).

3.1 Preliminaries

First, we examine the problem of visualising answer sets. It is arguably a hard task to evaluate and analyse answer sets as output by a solver. Consider, e.g., the output of an encoding of a maze generation problem. Here, a maze is a two-dimensional grid, where every cell contains either a wall or is empty and it has exactly one entrance and one exit. An answer set of a maze encoding as output of a solver could be as follows:

```
entrance(1,2) exit(15,9) row(1) row(2) row(3) row(4) row(5) row(6) row(7)
row(8) row(9) row(10) row(11) row(12) row(13) row(14) row(15) col(1) col(2)
col(3) col(4) col(5) col(6) col(7) col(8) col(9) col(10) col(11) col(12)
col(13) col(14) col(15) border(1,15) border(1,14) border(1,13) border(1,12)
...
wall(13,15) wall(14,15) wall(15,2) wall(15,3) wall(15,4) wall(15,5)
wall(15,6) wall(15,7) wall(15,8) wall(15,10) wall(15,11) wall(15,12)
wall(15,13) wall(15,14) wall(15,15) wall(2,1) wall(3,1) wall(4,1)
...
adjacent(1,7,1,8) adjacent(1,8,1,9) adjacent(1,9,1,10) adjacent(1,10,1,11)
adjacent(1,11,1,12) adjacent(1,12,1,13) adjacent(1,13,1,14)
adjacent(1,14,1,15) adjacent(2,1,2,2) adjacent(2,2,2,3) adjacent(2,3,2,4)
adjacent(2,4,2,5) adjacent(2,5,2,6)
...
reach(13,13) reach(14,12) reach(13,14) reach(14,14) empty(1,2) empty(2,2)
empty(2,4) empty(2,5) empty(2,6) empty(2,7) empty(2,8) empty(2,10)
empty(2,12) empty(2,14) empty(3,2) empty(3,4) empty(3,6)
```

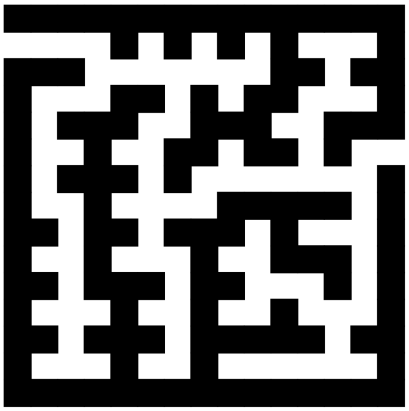

Figure 3.1: Example for a user-friendly visualisation of a maze.

This is only a short extract of the whole answer set. Clearly, one will agree that this is very cumbersome for the user to interpret. Our aim is to look for some visualisation that helps a programmer to better interpret output like this. An example for a possible visualisation of a maze problem can be found in Figure 3.1.

A further goal is that also people inexperienced with ASP should be able to interpret the visualisations of answer sets. To reach this goal, an abstraction has to be made between answer-set programming itself and the visualisation of the answer set.

Generally, two approaches how to visualise answer sets can be identified:

1. predicates and individuals of the original interpretation are mapped to visualisation predicates; and

2. a generic graph-like representation of the interpretation is employed.

The advantage of the first approach (like seen in Figure 3.1) is that this type of visualisation can be easily interpreted, because it displays the solution to the problem as natural as possible. The second type of visualisation can be helpful for expert answer-set programmers but novice ASP users may not easily understand this kind of visualisation. But there is also a difference in the visualisation process between these two approaches: While the latter one can be achieved without any user input to the visualisation, the former one needs information about the visualisation by the user. This information must include which predicates and individuals may be displayed as which graphical elements. E.g., if the original interpretation contains a predicate $book/1$, the user must give the visualisation a "hint" that the book should be displayed as a rectangle, otherwise the visualisation component cannot know how to display a predicate called "book". The best way to define such mappings from the predicate symbols of the domain to components of the visualisation is to use some kind of declarative language, like XML[1] or even ASP itself. Note that also the generic approach can rely on user input for knowing what to render as nodes and edges. This is also domain-specific as done in `Alloy` [25].

[1] "XML" stands for "eXtensible Markup Language".

In Kara, both ways of visualisation were implemented. For the former approach, ASP itself is used as the host language. A so-called *visualisation program* V is defined, which includes rules to map predicate symbols of the original interpretation to special visualisation predicates from a set $\mathcal{P}_v$. Afterwards, these visualisation predicates are post-processed from the visualisation interpretation I_v and then the visualisation is rendered on the screen.

Problematic with current visualisation tools was the visualisation of problems relying on grids and the even more difficult problem was the visualisation of graph structures. Kara undergoes this problem with supporting the definition of grids inside its visualisation language as well as automatically rendering graphs defined in its visualisation language with the built-in graph-layouting algorithm of the Draw2d library of the Eclipse Graphical Editor Framework. This is a new feature currently not implemented in other tools, which offers a powerful method for visualising generic graph structures as well as easily render grids (grids can, e.g., be used for rendering a Sudoku puzzle[2]).

Offering these two types of visualisation should improve developing answer-set programs. However, this is only a first approach and it must be evaluated by programmers in practice. It has to be adapted in future to fully fit the answer-set programmers needs.

Eclipse

The Eclipse platform is a general tool providing integrated development support for programmers. The platform itself is written in the Java programming language and provides the possibility to extend its features through a plug-in system. Originally, Eclipse was introduced for Java developers, but today due to the built-in plug-in system, there are much more languages supported, like C, C++, Javascript, etc. There are also many tools trying to support the programmer like graphical representations of the code, test-coverage tools, graphical editors, and much more. Furthermore, there are also tools which support the programmer with offering a graphical user interface for versioning tools as well as ticketing systems.

The built-in plug-in system of Eclipse allows to extend the features of the platform as well as to add new features (e.g., add support for other programming languages or tools for still existing languages). All elements of the entire Eclipse platform can be reused (e.g., editors, markers, syntax highlighting, file comparisons, edit/paste, etc.). In contrast to writing Eclipse plug-ins, it is also possible to write standalone applications using the components of the Eclipse platform which is called *RCP* (*Rich Client Platform*). The whole plug-in system of Eclipse is referred to as *PDE* (*Plug-in Development Environment*) and features plug-in development as well as RCP.

Moreover, Eclipse features also a so-called *run-configurations environment* which is a very important feature for our work. It allows to configure settings for execution of different programs written by a software developer. For instance, consider a command-line program taking several options as input. These command-line options can be configured via the run-configurations environment and later on the program can be executed with only one click. This environment can also be used to configure different settings on the compiler, virtual machine, or the system environment variables.

[2] Sudoku is a number puzzle, where the numbers must be assigned to grids according to special rules.

Writing a new plug-in can be characterised by the following two components:

- a `manifest` and `plugin.xml` file specifying build properties and plug-in dependencies; and
- an activator class for plug-in initialising.

When these two components are created, the programmer can start with plug-in development. Dependencies defined in the `plugin.xml` file can either be components of the core system or other plug-ins. All components which can be used as dependencies in the same plug-in have their own conventions and thus the manual of the dependencies may be used how to employ those components.

SeaLion

`SeaLion` [27] is a an `Eclipse` plug-in aiming to provide an *integrated development environment* (*IDE*) for answer-set programming. It is the *base plug-in* for all other components inside the answer-set programming IDE and thus offering the basic functionality of the IDE. The features of the `SeaLion` IDE include:

- semantic code highlighting in an editor of `Eclipse`;
- executing answer-set programs through `Eclipse` *run configurations*;
- parsing of the output of both `Clasp` as well as `DLV`; and
- *interpretation view* to show the solver output graphically in a tree representation.

The visualisation system introduced in this thesis depends on the `SeaLion` plug-in using all its basic feature like the editor with code highlighting, the interpretation view as well as the run-configurations environment. Moreover, `SeaLion` offers the possibility to launch a solver (e.g., `Gringo` or `DLV`) with a specific configuration and later on after the execution finishes receiving the result as a parsed set of `Java` objects. It is also possible to inject some configuration-specific changes before the answer-set program is executed by the solver. Furthermore, the *interpretation view* of `SeaLion` is used for choosing interpretations to visualise as well as to output abduced interpretations and visualisation interpretations.

Graphical Editing Framework (GEF)

The *Graphical Editing Framework* (*GEF*) [28] allows developers to create graphical editors for use within `Eclipse`. It is used in our work to provide the graphical representations of the answer sets as defined by the user as well as to provide editing possibilities to the visualisation. The two basic components of the framework used for the visualisation component are `Draw2d` and GEF.

`Draw2d` provides layout and rendering functionality on the basis of the *Standard Widget Toolkit* (*SWT*). In our approach, it is used to draw graphical objects like rectangles, circles, etc.

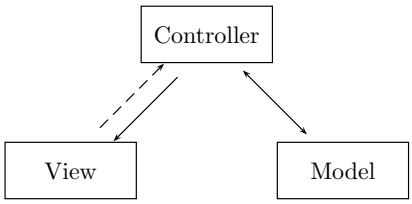

Figure 3.2: Model-View-Controller pattern as implemented in the Graphical Editing Framework.

The other part, GEF, is used to provide a *Model-View-Controller* (*MVC*) architecture. Figure 3.2 depicts the framework to organise the various used components.

In general, the model of GEF consists of *POJOs* (*Plain Old* `Java` *Objects*) which store the properties and relations between the objects in the graphical view. When the model is built, a so-called `EditPartFactory` is used to generate the corresponding controllers to the model objects based on the *factory pattern* for generating objects based on parameters in the factory. The controller objects belong to the class `EditPart` and are most of the time in a one-to-one relation to the model objects. Note that an `EditPart` object can itself contain other `EditPart` objects (e.g., a container object for figures like rectangles or ellipses). These controllers themselves construct the view objects depending on the information in the underlying model objects. In general, the view objects are figures depending on the `Draw2d` layout and rendering engine which can be custom compound objects or figures directly provided by `Draw2d`. Standard figures provided by `Draw2d` include, for instance, rectangles, ellipses, polygons, and images. Moreover, after the view objects are generated the user can see the graphical visualisation in the editor.

The next feature of the framework are the editing possibilities which can be used by adding objects from the class `EditPolicy`. They determine how the created objects can be edited and provide also callback functions for the controller which can be used to adapt the model, respectively.

Moreover, the framework also provides a feature allowing for saving the graphical representations in the editor, but this is not discussed in this work.

The `Kara` system

The goal of our work was to develop a visualisation component for answer-set programming, where the visualisation should be as close as possible to the actual problem. The idea is to translate an answer set of the program to an input format which can then be processed by the visualisation component. For this translation process, a declarative language has to be determined. As there are already two successful visualisation approaches available, namely `ASPVIZ` [22] and `IDPDraw` [21], which are using ASP itself to model the visualisation program, also our tool, `Kara`, uses ASP as visualisation language.

As stated above, in general, two answer-set programs are needed in order to visualise the original problem:[3]

[3]In fact, it is possible to make "stand-alone" visualisations too. This will be shown later on in Section 5.

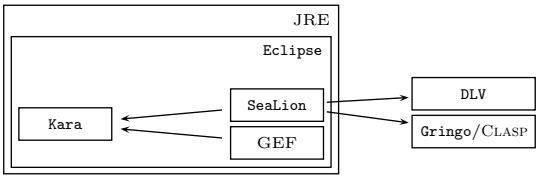

Figure 3.3: Technology stack of the Kara system.

- the original answer-set program; and
- a program mapping the original answer set to an input for the visualisation component.

As described above, the plug-in Kara is implemented within the Eclipse platform, which itself is executed inside a *Java Runtime Environment*. The technology stack of the Kara system is depicted in Figure 3.3. The tool was implemented with Java version 6 but should also be compatible with upcoming versions of it. Kara depends on two other Eclipse plug-ins, namely GEF and SeaLion. It also depends on external solvers like DLV and Clasp (in conjunction with Gringo). The external solvers are not all necessary, but at least one of them should be available on the user's system for solving answer-set programs. There can be also more external solvers but the parser for them is not implemented in the SeaLion plug-in yet.

Although Kara is implemented in Java and is executed within the Eclipse environment, its implementation contains also two answer-set programs, which are used to ease the implementation of the layout on the graphical editor. Currently, there is a so-called *positioning script* available for DLV as well as for Gringo (these are listed later on).

As it can be seen in Figure 3.3, the interface between Kara and the answer-set programming solvers is the SeaLion plug-in, which has an implemented parser. SeaLion executes the answer-set program with the configured solver and returns the answer-sets of the program as Java objects via a listener. Afterwards, the answer sets as well as all parts of them (atoms, literals, constants, function symbols) can be read, edited, and created via utility methods provided in SeaLion.

The graphical representation and editing features are done with the GEF plug-in of the Eclipse framework. Therefore, the MVC structure of the plug-in was implemented in the Kara system. The model is implemented inside the Kara system as an answer-set represented by a Java object in the SeaLion core. For every possible visualisation object in the model answer set, there exists a corresponding Java class in the model. For every object in the visualisation answer set, an instance of the corresponding Java class is created and the properties of each object is set accordingly to the values in the visualisation answer set.

Furthermore, there is an EditPart for every possible graphical object that is supported inside this plug-in which are the controllers in the framework. Then, the instances of the class EditPart create the corresponding graphical objects which are of type Figure but only standard figures of Draw2d are used. By reading the model properties, the controller sets these properties on the figures. Moreover, for positioning graphs, the layout algorithm of Draw2d is used.

Notation

Some definitions and naming conventions must be introduced in order to fully understand the technical content of the next chapters. If the name *domain program* is used, the "original" answer-set program is meant (i.e., the program the developer has written in order to solve a given problem) and its answer sets are denoted by $I_0, \ldots, I_n$. The *visualisation program*, V, is an answer-set program mapping an answer set $I \in \{I_0, \ldots, I_n\}$ of the domain program to a graphical representation with its corresponding interpretation I_v. This means that every predicate of the domain program should be assigned with an element in the visualisation. Of course, for a real-world representation of the problem, it can also be necessary to visualise elements which are not part of the answer set of the domain program (e.g., for a chessboard problem, using black and white fields). Predicates with special semantic meaning are introduced in order to know what the programmer wants to visualise which are called *visualisation predicates*. Furthermore, there exists also a subset $\mathcal{P}_i$ of $\mathcal{P}_v$ referred to as *integrity predicates* which are needed later when the abduction framework is introduced. Because we also want to edit interpretations, we use I' to denote the *modified interpretation* corresponding to the answer set I of the domain program. Accordingly, I'_v denotes the interpretation corresponding to the edited visualisation.

Definition of the syntax

As the prerequisites are now defined, i.e., choosing the technologies and frameworks behind the implementation, the next step is to define the syntax of the custom (reserved) visualisation predicates for obtaining the input for the visualisation component. The goal is to create "language elements", which allow to write short and effective visualisation programs. In general, three types of visualisation predicates with different meanings were introduced, namely

1. predicates for the definition of elements for the graphical visualisation,
2. predicates for setting properties for the graphical elements; and
3. predicates for defining properties which the user is allowed to change in the visualisation.

The first type of predicates allows the user to create elements in the graphical visualisation, where the mandatory properties are given as arguments. The second type of predicates, which are only optional, were introduced to minimise the configuration overhead for the user (e.g., most users only want to use black foreground- and white background colour). These predicates may only be used if the user wants to set optional properties to special values (e.g., setting the background colour to green). Last but not least, the third type of predicates characterises those predicates defining the changeable properties of each element. If there is no "changeable predicate" provided for an element, then no property of this element can be changed.[4] As this type of predicates are referred to the modification of the graphical representation, they are also used to create new elements, which defines an element to be able to be copied.

All visualisation predicates contain the prefix vis in order to distinguish them from the predicates of the domain program. For example, if the $colour$ predicate is used in the domain program, the user uses the $viscolour$ predicate in the visualisation program.

[4] Note that the position in the graphical editor is always changeable (i.e., x and y coordinates of the elements).

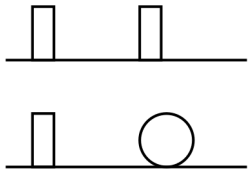

Figure 3.4: Custom visualisation of the book example.

For some argument values of visualisation predicates, fixed constant symbols are used (e.g., italics and bold font for the line style). If for those special arguments other values than the defined ones are used, the predicates are simply ignored.

Example 3.1. Assume we are given a domain program determining the position of books and globes on a shelf. The program has several answer sets, where one of them is the following:

$$I = \{book(s_1, 1), book(s_1, 3), book(s_2, 1), globe(s_2, 2)\}.$$

This is a text representation like we receive it from a solver (e.g., on the command line). Each of the constants s_1 and s_2 represents a shelf, and the atoms $book(X, Y)$ and $globe(X, Y)$ express that there is a book or globe in row Y of shelf X, respectively. To visualise this interpretation, a visualisation program can be written in order to "translate" the atoms of the domain program in some kind of visualisation. The visualisation program can look as follows:

$$visline(shelf_1, 10, 40, 80, 40, 0), \tag{3.1}$$
$$visline(shelf_2, 10, 80, 80, 80, 0), \tag{3.2}$$
$$visrect(f(X, Y), 20, 8) :- book(X, Y), \tag{3.3}$$
$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- book(s_1, Y), \tag{3.4}$$
$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- book(s_2, Y), \tag{3.5}$$
$$visellipse(f(X, Y), 20, 20) :- globe(X, Y), \tag{3.6}$$
$$visposition(f(s_1, Y), 20 * Y, 20, 0) :- globe(s_1, Y), \tag{3.7}$$
$$visposition(f(s_2, Y), 20 * Y, 60, 0) :- globe(s2, Y). \tag{3.8}$$

It is easy to see that predicates of the interpretation of the domain program are mapped to visualisation predicates. In the first two rules, two shelfs are generated, where the books and globes are placed. The predicate $visline$ takes the identifier of the element, the (x, y) position of the starting and end point, as well as the z-coordinate as arguments. The z-coordinate is used to determine which element should be rendered if two of them are overlapping. If both elements have the same z-coordinate, the behaviour of the visualisation is undefined.

Books are represented as rectangles with a width of 20 and a height of 8 and their identifier is dynamically generated by using a function symbol $f(X, Y)$, where X and Y refer to the logical position in the domain program. Function symbols have no processing functionality, they are only used for generating a new individual in the domain by taking some arguments. With the predicate $visposition$, the rectangles (i.e., the books) can be placed on the canvas. Shelf 1 has y-coordinate 20, whereas Shelf 2 has y-coordinate 60. The x-coordinate is calculated dynamically, where a size of 20 pixels is taken per element (i.e., for globe or book).

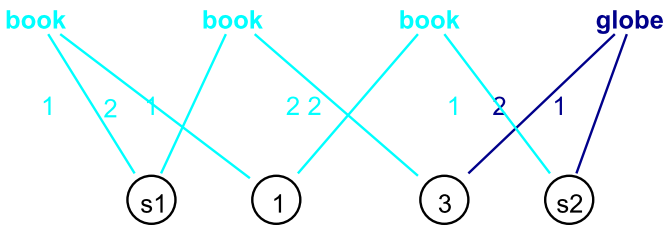

Figure 3.5: Generic visualisation of the book example.

Rules (3.6)-(3.8) are used to render globes on the canvas. It is quite the same as for books, with the difference that globes are drawn as circles instead of rectangles.

Important is that the identifiers of the elements are used to refer to this element in the whole visualisation program. Every element gets a unique identifier in the visualisation program. E.g., we use this identifier for rectangles and circles to set the their position on the canvas later on. It is also possible to use this identifier to set more properties on the elements like colour, line style, and many more. Elements are *not* allowed to have the same identifier, because then the correct intention of the programmer cannot be guaranteed (i.e, to assign the specified property to the intended element). Therefore, created or copied elements also get a new identifier by `Kara`.

The visualisation of this example is depicted in Figure 3.4. ♢

Example 3.2. Another possibility for visualisation would be the generic visualisation feature of `Kara`. This kind of visualisation yields a graph-like visualisation, where the constants (i.e., s_1, s_2, 1, and 3) are rendered as nodes marked as circles. For every predicate (i.e., $book$ and $globe$), a simple text node is rendered. Constants are rendered only once for the whole visualisation program, whereas predicates are rendered per occurrence. Every predicate and constant which belongs to each other are connected via straight lines, where the label of the line belongs to the argument count of the constants inside the predicate. The generic visualisation of the book example can be found in Figure 3.5. ♢

3.2 System overview

This chapter explains the workflow and the architecture of the overall system in detail. Besides this rather abstract system description, a technology overview is given too.

Workflow

The overall workflow is given in Figure 3.6. Assume that there already exists some domain program Π, which is the encoding of a problem which should be solved. Joined with some input and executed by a solver, the output can comprise several answer sets. Afterwards, the user chooses one of these answer sets, I, for the visualisation and writes a visualisation program, V, which is responsible for the mapping between the predicates of Π and their corresponding graphical visualisation. Thus, dedicated visualisation predicates are defined, which define the graphical elements as well as their properties (e.g., $visrect/3$, $viscolor/2$, ...). The interpretation I is the

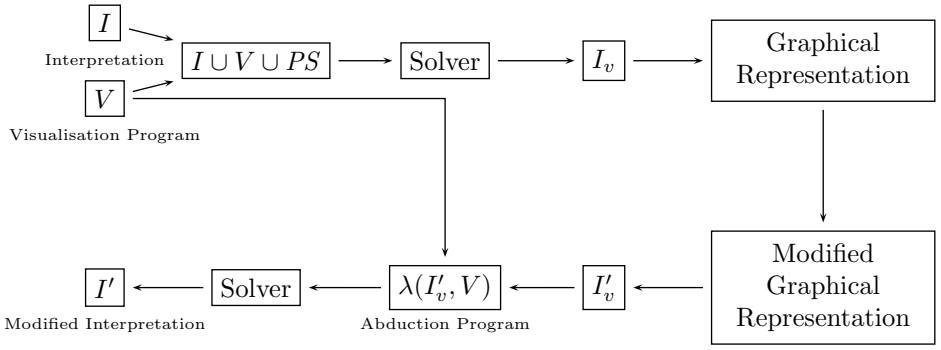

Figure 3.6: Overall workflow of the visualisation system.

input of V and $V \cup I \cup PS$ is executed by a solver, which outputs then the visualisation answer set I_v which is filtered for the visualisation predicates $\mathcal{P}_v$. The program PS is the *positioning script* and is responsible for the layout if the user takes advantage of relative positioning features in his visualisation program. Currently, `Kara` includes two positioning scripts, one for `DLV` and one for `Clasp`, which can be chosen by the user when making the visualisation. Then, it is used by `Kara` to render the desired graphical representation of the problem. Now the user has the possibility to graphically modify the visualisation in the editor pane of `Eclipse`. Afterwards, the corresponding interpretation I' for the modified visualisation interpretation I'_v is inferred. Because the encoding of I'_v contains only visualisation and auxiliary predicates, it can be necessary to compute the interpretation I' with the original encoding of the domain program. For inferring I', an abduction program $\lambda(I'_v, V)$ is generated by `Kara`, which itself is also an answer-set program and takes as input the graphically modified visualisation interpretation I'_v as well as the visualisation program V. If the abduction program is executed, it outputs I' such that $I' \cup V$ yields the modified visualisation interpretation I'_v.

Visualisation

As stated in the introduction, the visualisation approach of `Kara` follows the methods of the previous systems `ASPVIZ` and `IDPDraw`. The general idea of the visualisation is as follows: Given an interpretation I of a domain program P defined over a first-order alphabet $\mathcal{A}$, I is joined with the visualisation program V which is written by the user and defined over a first-order alphabet $\mathcal{A}' \supset \mathcal{A}$. $\mathcal{A}'$ contains the predicates of P, function symbols, as well as further visualisation predicates from a set $\mathcal{P}_v$ and some optional auxiliary predicates if needed. $\mathcal{P}_v$ is a set of fixed visualisation predicates, which are needed to define graphical elements like rectangles, ellipses, etc. as well as their properties like background colour, position, and so on. An exhaustive list of available visualisation predicates is given in Table 3.1. V, which is itself an answer-set program, is used to define a mapping of I to some corresponding visualisation. Thus, the user defines rules which generate for predicates contained in I some visualisation predicates from $\mathcal{P}_v$. For instance, a simple rule could be the following:

$$visrect(f(X,Y),20,20) :- wall(X,Y).$$

Atom	**Intended meaning**
$visellipse(id, height, width)$	Defines an ellipse with specified height and width.
$visrect(id, height, width)$	Defines an rectangle with specified height and width.
$vispolygon(id, x, y, ord)$	Defines a point of a polygon. The ordering specifies in which order the defined are points are connected with each other.
$visimage(id, path)$	Defines the image given in the specified file.
$visline(id, x_1, y_1, x_2, y_2, z)$	Defines a line between the points (x_1, y_1) and (x_2, y_2).
$visgrid(id, rows, cols, h, w)$	Defines a grid, with the specified number of rows and columns. Height and width define the size of the grid.
$visgraph(id)$	Defines a graph.
$vistext(id, text)$	Defines a text element.
$vislabel(id_g, id_t)$	Sets the text element id_t as a label for graphical element (id_g). Labels are supported for the following elements: $visellipse/3$, $visrect/3$, $vispolygon/4$, and $visconnect/3$.
$visisnode(id_n, id_g)$	Adds the graphical element id_n as a node to a graph id_g for automatic layouting. The following elements are supported as nodes: $visrect/3$, $visellipse/3$, $vispolygon/4$, and $visimage/2$.
$visscale(id, height, weight)$	Scales an image to the specified height and width.
$visposition(id, x, y, z)$	Puts an element id on the fixed position (x, y, z).
$visfontfamily(id, ff)$	Sets the specified font ff for a text element tt.
$visfontsize(id, size)$	Sets the font size $size$ for a text element tt.
$visfontstyle(id, style)$	Sets the font style for a text element tt to bold or italics.
$viscolor(id, color)$	Sets the foreground colour for the element id.
$visbackgroundcolor(id, color)$	Sets the background colour for the element id.
$visfillgrid(id_g, id_c, row, col)$	Puts element id_c in cell (row, col) of the grid id_g.
$visconnect(id_c, id_{g_1}, id_{g_2})$	Connects two elements, id_s and id_t, by a line such that id_s is the source and id_t is the target of the connection.
$vissourcedeco(id, deco)$	Sets the source decoration for a connection.
$vistargetdeco(id, deco)$	Sets the target decoration for a connection.
$visleft(id_l, id_r)$	Ensures that the x-coordinate of id_l is less than that of id_r.
$visright(id_r, id_l)$	Ensures that the x-coordinate of id_r is greater than that of id_l.
$visabove(id_t, id_b)$	Ensures that the y-coordinate of id_t is smaller than that of id_b.
$visabove(id_t, id_b)$	Ensures that the y-coordinate of id_t is smaller than that of id_b.
$visinfrontof(id_1, id_2)$	Ensures that the z-coordinate of id_1 is greater than that of id_2.
$vishide(id)$	Hides the element id.
$visdeletable(id)$	Defines that the element id can be deleted in the visual editor.
$viscreatable(id)$	Defines that the element id can be created in the visual editor.
$vischangable(id, prop)$	Defines that property $prop$ can be changed for element id in the visual editor.
$vispossiblegridvalues(id, id_e)$	Defines that graphical element id_e is available as possible grid value for a grid id in the visual editor.

Table 3.1: Predefined visualisation predicates and their intended meaning.

This rule of V would generate a rectangle with a width and height of 20 pixels for every wall of the input interpretation I. The output of $V \cup I \cup PS$, the visualisation interpretation I_v, is then filtered for the visualisation predicates $\mathcal{P}_v$ and is post-processed by `Kara` to produce the graphical representation.

First, a super element, `InterpretationEditPart`, is constructed where all other graphical elements defined by the visualisation answer set are placed on. This element is a simple panel containing the other elements as children. For constructing and adding the children to the `InterpretationEditPart`, Algorithm 1 is used. The atoms of I_v are converted to model beans containing the data of the graphical elements which should be rendered. First, all graphical elements of the visualisation interpretation are collected, which are text elements, rectangles, images, ellipses, lines, grids, polygons, graphs, as well as connections between elements in a graph. We use the method $getIds()$ to get the identifiers of a specific predicate in some interpretation. Afterwards, all elements which are either hidden or a child element of another element must be removed from the set $Visible$ of visible elements. Hidden elements can be used for the creation of new elements in order to modify the graphical representation whereas child elements must only be rendered by their parent elements. By using the predicate $vishide/1$, hidden elements can be defined, whereas elements referenced by $vislabel/2$, $visisnode/2$, $visfillgrid/4$, or $visconnection/2$ are used as child elements of their parent container. The parent element of a label can be a rectangle, a polygon, an ellipse, or a connection, whereas the parent element of a node or a connection is the graph while the parent element of a grid cell it is the grid itself. All connections and graphs are also needed separately to calculate the graph layout and set the target and source connections on all elements connected to each other. Furthermore, we also need to set all collected elements as a property to every visible element, because, e.g., otherwise a rectangle would not be able to render its label. After setting target and source connections on all elements as well as calculating the graph layout for each graph, Algorithm 2 is used in order to calculate the absolute position of relatively positioned elements. Because the graph layout was already calculated and the absolute positions were stored in the properties of each node, the graph elements must be removed from the set of visible elements and the nodes must be added in order to be rendered on the editor pane. Finally, the set of visible elements is returned by the procedure.

Algorithm 2 is used for absolutely layout relatively positioned elements. The predicate $vislpos/2$ is used by `Kara` to store the relative position of elements. We need to determine the set of all $vislpos/2$ atoms of the interpretation in order to convert them to absolute positions. We put all $vislpos/2$ literals in a map called $Figures$ which is sorted accordingly to their y, x as well as z position. Furthermore, we define two variables, one for storing the x-positions of the elements and one for storing the y-positions. First we initialise these variables with the first values of the first figure. Additionally we need to store the maximal y value for every row we have in the logical positions done with $yMax$ variable. Next, we iterate over all figures and calculate their absolute positions. Due to the fact that the figures are sorted accordingly to their positions we only have to insert them after the last inserted figure. In the end we store the calculated absolute positions directly in the figures which are later on displayed on the screen.

Another feature following from this approach is that one can easily create scalable vector graphics (SVG) by defining an answer-set program with facts what to draw. This is a much

Algorithm 1 Algorithm for converting atoms of I_v to graphical elements in the editor pane.

Input: interpretation I_v
Output: a collection of visible elements $visibleElems$

$Visible \leftarrow getIds(vistext) \cup getIds(visrect) \cup getIds(visimage) \cup getIds(visellipse) \cup$
$getIds(visline) \cup getIds(visgrid) \cup getIds(vispolygon) \cup$
$getIds(visgraph) \cup getIds(visconnect)$
$Visible \leftarrow Visible \setminus getIds(vishide) \cup getIds(vislabel) \cup getIds(visisnode) \cup$
$getIds(visfillgrid) \cup getIds(visconnection)$
$Connections \leftarrow getIds(visconnect)$
$Graphs \leftarrow getIds(visgraph)$
for all $Connection \in Connections$ **do**
set target connection in $Connection.target$
set source connection in $Connection.source$
end for
add all collected elements (including the hidden ones) to every element
for all $Graph \in Graphs$ **do**
calculate graph-layout for $Graph$
end for
call Algorithm 2 for positioning relative elements
$Visible \leftarrow Visible \setminus Graphs$
$Visible \leftarrow Visible \cup Nodes$
return $Visible$

easier approach than to write plain XML. Furthermore, rules can be defined which can generate an element more than once with one simple statement.

Kara supports also generic visualisations in which the input interpretation I is represented as a hypergraph without any user input. Thus, no visualisation program is needed in this case. The construction of such hypergraphs is determined by Algorithm 3. In this method, every occurrence of a predicate is converted to a node in the visualisation interpretation. Furthermore, every predicate is assigned a colour, where equal predicate names have also equal colour. Every constant and function symbol is also converted as a node in the visualisation interpretation. The only difference to the rendering of predicates is that the constant symbols are rendered uniquely, which means that if one constant occurs multiple times in the input interpretation I, it is rendered only once in the output. If a constant symbol occurs in a specific predicate, it is connected with this predicate only via an edge in the hypergraph. The label of the edge is the location of the constant symbol inside the predicate and the edges are assigned the same colour as their corresponding predicate symbol.

Relative positioning and higher level elements

Kara supports special purpose elements which allow for writing shorter as well as easier visualisation programs. They are used to release the programmer of the burden of the layouting problem of the visualisation. Additionally, Kara supports the use of relative positioning to

Algorithm 2 Algorithm for converting the relative position to an absolute one.

Input: visualisation interpretation I_v
Output: the absolute positions are set for every figure of the input interpretation I_v

$Lpos \leftarrow$ get all $vislpos$ predicates from I_v
$Figures \leftarrow \emptyset$
for all $lit \in Lpos$ **do**
  $Figures.put(lit.position, lit.element)$
end for
sort $Figures$ according to y, x, z coordinates ascending
$Xpositions.put(Figures[1].key.x, \{0, Figures[1].value.width\})$
$Ypositions.put(Figures[1].key.y, 0)$
$yMax \leftarrow Figures[1].value.height$
for $i = 2$ **to** $Figures.size$ **do**
  $figure \leftarrow Figures[i]$
  if y coordinate changed **then**
    $Ypositions.put(figure.key.y, yMax)$
  end if
  $start \leftarrow$ get starting position for this element
  if $figure.key.x \notin Xpositions$ **then**
    $Xpositions.put(figure.point.x,$
      $\{start.value.pos + start.value.width, figure.value.width\})$
    update all positions of elements after this element
  else if $Xpositions[figure.key.x].width < figure.value.width$ **then**
    $Xpositions.put(figure.point.x, \{start.value.pos, figure.value.width\})$
    update all positions of elements after this element
  end if
  $yMax \leftarrow maximum(yMax, Ypositions[figure.key.y] + figure.value.width)$
end for
store calculated positions in figures

make drawing problems easier, which only rely on relative positions instead of absolute ones.

The first feature to mention is the graph layout. Users only have to define the graph via dedicated input predicates and `Kara` is responsible to produce a good layout of the graph on the editor pane. Internally, this is delegated to the graph-layout algorithm of the `Draw2d` library, where the corresponding `Java` class is called `DirectedGraphLayout`. This class takes as input the graph via the `Node` and `Edge` classes of the framework. Afterwards, with a simple call of the static `visit` method of the `DirectedGraphLayout` class, all absolute positions are stored in the data model (i.e., in the nodes and edges). Furthermore, also very good bendpoints[5] are calculated.

`Kara` also supports an element which uses a grid layout for its contained elements. The user defines a grid with a column and row size as well as a height and width in pixel. Then,

[5] A bendpoint is a corner of an edge, which is often used for the sake of readability.

Algorithm 3 Algorithm for constructing a hypergraph from the input interpretation I.

Input: interpretation I
Output: visualisation interpretation $Graph$

```
Graph ← ∅
for all Pred ∈ I.predicates do
  Colour ← assignColour(Pred)
  Graph ← Graph ∪ constructPredNode(Pred, Colour)
  for Term ∈ Pred.terms do
    if Term ∉ Graph then
      Graph ← Graph ∪ constructTermNode(Term)
      Graph ← Graph ∪ constructConnection(Pred, Term, Colour)
    end if
  end for
end for
return Graph
```

there is the possibility to fill the grid with a special predicate, where the user may only mention the element to position in the grid as well as the (x, y)-coordinates inside the grid (i.e., the column and the row). Grids can be very useful to later infer the corresponding interpretation to a modified visualisation and ease the positioning. Examples for useful grid problems are Sudoku, 15-puzzle, the social golfer problem, as well as the n-queens problem.

The relative positioning is done via special predicates which define the position of an element relative to another one. Elements can be on the left or on the right side of other elements, or they can be above or below other elements. `Kara` also supports an z-axis and thus offers the possibility for elements to be *in front of* other elements. The implementation of the relative positioning is a mix of an answer-set program as well as a specific `Java` layout algorithm. To set out the elements logically, an answer-set program is used. In the implementation, there are two of those: one program for `DLV` and the other one for `Gringo`/`Clasp`. The user provides as input the visualisation program V as well as the interpretation I to be visualised. Furthermore, also a layouting program needs to be chosen, either for `DLV` or for `Gringo`, depending on the preferred solver. The respective programs are depicted in Figures 3.7 and 3.8, respectively.

The so-called *positioning script* defines a logical grid and has rules which identify the elements needing relative positioning. These are all elements without any $visposition/4$ predicate and all $visline/6$ predicates. Then, there is a rule which guesses a logical position for all relatively positioned elements. Furthermore, there are constraints enforcing that every element must have exactly one position and that two different elements must not be at the same position. Finally, there are four constraints that ensure that the positioning constraints of the visualisation program are fulfilled (i.e., involving the predicates $visleft$, $visright$, $visbelow$, and $visabove$).

In the `DLV` script, Rule (3.9) provides the definition of the logical grid which has 15 columns and 15 rows. Every cell of the logical grid can contain a graphical element. Thus, normally there is more than one solution for positioning the elements on the grid. Rules (3.10) to (3.15) are responsible for determining every element of the graphical representation which have a fixed

$$vislogicgrid(15,15). \tag{3.9}$$

$$visfixed(X) :- visline(X,\_,\_,\_,\_). \tag{3.10}$$

$$visfixed(X) :- visposition(X,\_,\_,\_). \tag{3.11}$$

$$visfixed(X) :- vishide(X). \tag{3.12}$$

$$visfixed(X) :- visisnode(X,\_). \tag{3.13}$$

$$visfixed(X) :- visfillgrid(\_,X,\_,\_). \tag{3.14}$$

$$visfixed(X) :- vislabel(\_,X). \tag{3.15}$$

$$visshow(X) :- visgraph(X), \text{not } visfixed(X). \tag{3.16}$$

$$visshow(X) :- visgrid(X,\_,\_,\_,\_), \text{not } visfixed(X). \tag{3.17}$$

$$visshow(X) :- visrect(X,\_,\_), \text{not } visfixed(X). \tag{3.18}$$

$$visshow(X) :- visellipse(X,\_,\_), \text{not } visfixed(X). \tag{3.19}$$

$$visshow(X) :- vispolygon(X,\_,\_,\_), \text{not } visfixed(X). \tag{3.20}$$

$$visshow(X) :- visimage(X,\_), \text{not } visfixed(X). \tag{3.21}$$

$$visshow(X) :- vistext(X,\_), \text{not } visfixed(X). \tag{3.22}$$

$$vislpos(N,X,Y,0) \vee -vislpos(N,X,Y,0) :- visshow(N), vislogicgrid(V,W), X <= V, Y <= W, \#int(X), \#int(Y), \text{not } visfixed(N). \tag{3.23}$$

$$vislpos(N,0,0,0) :- \#count\{N : visshow(N)\} <= 1, visshow(N), \text{not } visfixed(N). \tag{3.24}$$

$$:- \#count\{X,Y,Z : vislpos(N,X,Y,Z)\} < 1, visshow(N), \text{not } visfixed(N). \tag{3.25}$$

$$:- \#count\{X,Y,Z : vislpos(N,X,Y,Z)\} > 1, visshow(N). \tag{3.26}$$

$$:- vislpos(N_1,X,Y,Z), vislpos(N_2,X,Y,Z), N_1 != N_2. \tag{3.27}$$

$$:- visleft(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,X_1,\_,\_), vislpos(N_2,X_2,\_,\_), X_1 >= X_2, visshow(N_2), vislpos(N_1,X_1,\_,\_). \tag{3.28}$$

$$:- visright(N_1,N_2), visshow(N_1), vislpos(N_2,X_2,\_,\_), X_2 >= X_1. \tag{3.29}$$

$$:- visabove(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,\_,Y_1,\_), vislpos(N_2,\_,Y_2,\_), Y_1 >= Y_2. \tag{3.30}$$

$$:- visbelow(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,\_,Y_1,\_), vislpos(N_2,\_,Y_2,\_), Y_2 >= Y_1. \tag{3.31}$$

Figure 3.7: Positioning script for DLV.

position. These elements need not be positioned by the script because it is already done so by the user. Therefore, we use the predicate $visfixed/1$ to mark these elements as fixedly positioned. Furthermore, we use the predicate $visshow/1$ to mark all elements which we want to layout relatively to each other. Rules (3.16) to (3.22) are responsible for this task. Rule (3.23) is disjunctive to guess a position in the logic grid for each element marked with the predicate $visshow/1$ and Rule (3.24) is used if there is only one element for positioning available. Constraints (3.25) and (3.26) make sure there is exactly one logical position for each element. Constraint (3.27) in turn is responsible for checking that no two elements are positioned on the same cell of the logical grid. Finally, Constraints (3.28) to (3.31) are used to make sure that the elements are correctly aligned according to the relative positioning predicates $visleft$, $visright$, $visbelow$, and

$$visgridposX(0..15). \tag{3.32}$$
$$visgridposY(0..15). \tag{3.33}$$
$$visfixed(X) :- visline(X,\_,\_,\_,\_,\_). \tag{3.34}$$
$$visfixed(X) :- visposition(X,\_,\_,\_). \tag{3.35}$$
$$visfixed(X) :- vishide(X,\_,\_,\_). \tag{3.36}$$
$$visfixed(X) :- visisnode(X,\_). \tag{3.37}$$
$$visfixed(X) :- visfillgrid(\_,X,\_,\_). \tag{3.38}$$
$$visfixed(X) :- vislabel(\_,X). \tag{3.39}$$
$$visshow(X) :- visgraph(X), \text{not } visfixed(X). \tag{3.40}$$
$$visshow(X) :- visgrid(X,\_,\_,\_,\_), \text{not } visfixed(X). \tag{3.41}$$
$$visshow(X) :- visrect(X,\_,\_), \text{not } visfixed(X). \tag{3.42}$$
$$visshow(X) :- visellipse(X,\_,\_), \text{not } visfixed(X). \tag{3.43}$$
$$visshow(X) :- vispolygon(X,\_,\_,\_), \text{not } visfixed(X). \tag{3.44}$$
$$visshow(X) :- visimage(X,\_), \text{not } visfixed(X). \tag{3.45}$$
$$visshow(X) :- vistext(X,\_), \text{not } visfixed(X). \tag{3.46}$$
$$vislpos(N,X,Y,0) :- vismoreThanOneShow, visshow(N), visgridposX(X), visgridposY(Y), \text{not } visfixed(N), \text{not } -vislpos(N,X,Y,0). \tag{3.47}$$
$$-vislpos(N,X,Y,0) :- vismoreThanOneShow, visshow(N), visgridposX(X), visgridposY(Y), \text{not } visfixed(N), \text{not } vislpos(N,X,Y,0). \tag{3.48}$$
$$vismoreThanOneShow :- visshow(X), visshow(Y), X! = Y. \tag{3.49}$$
$$vislpos(N,0,0,0) :- visshow(N), \text{not } visfixed(N), \text{not } vismoreThanOneShow. \tag{3.50}$$
$$:- visshow(N), vislpos(N,X_1,Y_1,Z_1), vislpos(N,X_2,Y_2,Z_2), X_1! = X_2. \tag{3.51}$$
$$:- visshow(N), vislpos(N,X_1,Y_1,Z_1), vislpos(N,X_2,Y_2,Z_2), Y_1! = Y_2. \tag{3.52}$$
$$:- visshow(N), vislpos(N,X_1,Y_1,Z_1), vislpos(N,X_2,Y_2,Z_2), Z_1! = Z_2. \tag{3.53}$$
$$:- visfixed(N), vislpos(N,\_,\_,\_). \tag{3.54}$$
$$:- \text{not } 1\ \#count\{vislpos(N,X,Y,Z) : visshow(N)\}\ 1, visshow(N). \tag{3.55}$$
$$:- vislpos(N_1,X,Y,Z), vislpos(N_2,X,Y,Z), N_1! = N_2. \tag{3.56}$$
$$:- visleft(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,X_1,\_,\_), vislpos(N_2,X_2,\_,\_), X_1 = X_2. \tag{3.57}$$
$$:- visright(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,X_1,\_,\_), vislpos(N_2,X_2,\_,\_), X_2 >= X_1. \tag{3.58}$$
$$:- visabove(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,\_,Y_1,\_), vislpos(N_2,\_,Y_2,\_), Y_1 >= Y_2. \tag{3.59}$$
$$:- visbelow(N_1,N_2), visshow(N_1), visshow(N_2), vislpos(N_1,\_,Y_1,\_), vislpos(N_2,\_,Y_2,\_), Y_2 >= Y_1. \tag{3.60}$$

Figure 3.8: Positioning script for `Gringo/Clasp`.

$visabove$.

We provide a second script for Gringo/Clasp because the syntax and meanings of certain concepts are sometimes a bit different than for DLV (e.g., concerning aggregate functions). The structure of the script is quite the same; in the following we give a short description.

In this script, we use two different predicates for representing the logical grid, namely $visgridposX/1$ as well as $visgridposY/1$, which represent the possible x and y coordinate (Facts (3.32) and (3.33)). Thus, we have also here 15 rows and 15 columns to align the elements. The next rules are the same as for the DLV script where we mark the elements as fixedly positioned with $visfixed/1$ (Rules (3.34) to (3.39) or as relatively positioned with $visshow/1$ (Rules (3.40) to (3.46)). Rules (3.47) and (3.48) are used to make the disjunction like in the DLV script with default negation and to guess a cell for each relatively positioned element. Next, Rule (3.49) sets the auxiliary predicate $vismoreThanOneShow/0$ and checks whether there is more than one element marked with $visshow/1$ (in the DLV script, we use the $count$ aggregate for this purpose). Rule (3.50) is only used if there is only one element for positioning. Constraints (3.51) to (3.53) are used to make sure that a single element has exactly one position in the logic grid. Constraint (3.54), then, is used that no fixed elements get relatively positioned in the logic grid. Constraint (3.55) checks that there is for every $visshow/1$ element a logical position assigned in the grid, and Constraint (3.56) is used that not two different elements are assigned to the same position in the logical grid. Finally, Constraints (3.57) to (3.60) have the same purpose as in the DLV script.

Prior to execution with a solver, Kara internally builds $I \cup V \cup PS$, where PS stands for one of the positioning scripts. The output of the solver can now comprise several answer sets, which contain the position information in their encoding. Now, one answer set is selected and post-processed by Kara, where Algorithm 2 is used to determine suitable absolute positions for relative positioned elements. All logical position predicates $vislpos/4$ are extracted from the visualisation answer set and the mapping between the logical position of the element and the figures is stored in $Figures$, which is sorted first by the logical y-coordinate, second by the x-coordinate, and lastly by the z-coordinate of the corresponding figures. $Xpositions$ and $Ypositions$ are used to map the logical (x, y) position of each element to the absolute one. The layout is constructed like a grid. For every grid position, the element with the largest width determines the width of the cell in the grid. Furthermore, for every element, the starting x position is determined, which can be constructed by the starting position of the preceding element as well as the width of the preceding element. The cell of the current element is then set to the new width, if there was not already an element at this grid cell or the element has a bigger width than all other elements inserted before in this cell. If the width of a grid cell is changed, the positions of all other elements following the current element must be updated with the difference of the old width and of the width of the current element. Determining the y-coordinate for the current cell is straightforward, because $Figures$ is sorted according to the y-coordinate. For every row, the maximal y-coordinate is calculated (according to the starting position of the row and the maximal height of an element in this row) and set after the y-coordinate changed. In the end, the calculated positions must be stored in the graphical elements properties.

Editing

Once the visualisation is done, the user may want to edit the visualisation, which can have two reasons:

- the visualisation is incorrect; or
- generating different interpretations.

If the visualisation is incorrect, the user may want to debug the answer-set program. There already exist several debugging approaches [19, 20, 10] where one needs the answer-set program P as well as an interpretation I as input. Then, these debugging methods provide reasons why the given interpretation I is not an answer set of the program P. Thus, someone may construct a correct interpretation from the incorrect one by graphically editing its visualisation and then use the constructed interpretation as input for a debugging tool. Of course, before the graphically modified interpretation I'_v (which contains only visualisation predicates) can be used as an input for a debugging tool, the corresponding interpretation I' must be determined from I'_v. This is non-trivial and corresponds to an *abduction task*, which is explained in detail in Section 3.2.

Another use case for editing the graphical visualisation is to generate answer sets for testing purposes. Consider, e.g., modular programming in ASP, where different modules depend on other modules. Assume that module A depends on the output of module B, which is not finished at the moment whereas A is already finished. In order to test module A, *mock answer sets* are needed, which can be generated by using `Kara`s editing feature.

In `Kara`, the editing of the graphical visualisation is implemented by using the Eclipse Graphical Editor Framework (GEF) [28]. GEF offers a rich framework for all editing purposes including the definition of actions as well as commands. Actions are, for instance, rendered in the context menu of the editor whereas commands correspond directly to the elements properties on the editor pane (e.g., size, position, creation of elements, etc.). Furthermore, every component (in GEF referred to as $EditPart$) has a method named $createEditPolicies$ which defines the properties and possibilities of modifying those components. These editing possibilities are activated by the occurrence of their corresponding predicates defined in `Kara`. All modifications the user executes on the editor pane offer a callback, which we use to keep the answer set behind the graphical representation consistent with the user interaction in the front-end.

Abduction

In the preceding section, we described what applications the visual editing of answer sets can have and how it is implemented in `Kara`. This section describes the theoretical details about the abduction problem of the visual editing.

After the user finished editing the visualisation, the modified visualisation interpretation I'_v is obtained from which we have to infer a corresponding interpretation I'. We make use of abductive reasoning techniques in order to solve this problem.

An abduction problem is defined by a logic theory T and an observation O [32]. The problem is to find a hypothesis H which explains O, that is such that

- $T \cup H$ is satisfiable and

- $T \models H \rightarrow O$.

In our case, the logic theory is the visualisation program V, the observation is the modified visualisation interpretation I'_v, and the desired hypothesis is a corresponding interpretation I'. The visualisation program comes as input from the user and the modified visualisation interpretation from Kara as it was kept consistent behind all user operations during the modifying process of the visualisation. To infer a corresponding interpretation I', Kara constructs an answer-set program $\lambda(I'_v, V)$, referred to as *abduction program*, which is itself also an answer-set program. The input of the abduction program is the modified visualisation interpretation (the observation) and the visualisation program (the logic theory), and the output (with projection to the visualisation predicates) is a corresponding interpretation I' (the hypothesis).

The first problem faced when inferring I' is that the domain of the predicates of the domain program P must be known for *rule safety*. A simple approach would be to collect all constants and function symbols of $I \cup V$ to obtain the domain, but this would be rather inefficient, because there can be many new constants and function symbols not needed for the abduction process (i.e., when using auxiliary predicates, constructing new function symbols from constants of I or deleting elements). Furthermore, if some elements are added to the visualisation interpretation, the abduction program would not work any more due to missing constants or function symbols. Thus, we chose another approach which constructs rules to compute the domain from the modified visualisation interpretation. Moreover, there are guessing rules in $\lambda(I'_v, V)$ which use the computed domain in order to guess some atoms of the domain program P, referred to as *abducible atoms*. These atoms are then used together with the visualisation program V to derive a hypothetical interpretation I''_v. We use two types of constraints to ensure that I''_v coincides with I'_v, namely:

- the modified visualisation answer set I'_v as constraints; and
- constraints to derive only atoms contained in I'_v.

The visualisation answer set is translated into constraints which ensure that all atoms contained in I'_v are also derived by the abduction program (e.g., from $visrect(rect, 20, 20)$, we derive $:- \text{ not } visrect(rect, 20, 20)$). Furthermore, we make use of negative constraints, which are used that we derive no visualisation atoms which are not contained in I'_v. For these constraints, we use only a special set of predefined predicates $\mathcal{P}_i \subset \mathcal{P}_v$ called *integrity predicates*. We did not want to use all of the visualisation predicates because some of them need a very high preciseness, which is often very hard and sometimes not possible to draw by hand in the graphical editor like absolute positioned elements (e.g., $visline/6$, $visposition/4$, ...).

Finally, by computing the answer set of the constructed answer-set program $\lambda(I'_v, V)$ and projecting it to the guessed atoms of the domain program P, we retrieve the modified interpretation I'. We assume that for each domain interpretation I, every pair I_1 and I_2 of answer sets of $V \cup I'$ do not differ projected to $\mathcal{P}_i$.

Now, the problem described above can be rephrased as follows:

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

$$
\begin{aligned}
\mathsf{dom}(I'_v, V) = \{&nonRecAbddom(t) :- vInVAS(\vec{t'}) \mid r \in V, v/m \in \mathcal{P}_v, v(\vec{t'}) \in \mathrm{H}(r),\\
&\quad a(\vec{t}) \in \mathrm{B}^+(r), \vec{t} = t_1, \ldots, t, \ldots, t_n, a/n \notin \mathcal{P}_v,\\
&\quad VAR(t) \neq \emptyset,\ VAR(t) \subseteq VAR(\vec{t'})\} \cup\\
&\{abddom(t) :- vInVAS(\vec{t'}), nonRecAbddom(X_1), \ldots, nonRecAbddom(X_l) \mid\\
&\quad r \in V, v/m \in \mathcal{P}_v, v(\vec{t'}) \in \mathrm{H}(r), a(\vec{t}) \in \mathrm{B}^+(r), \vec{t} = t_1, \ldots, t, \ldots, t_n,\\
&\quad a/n \notin \mathcal{P}_v,\ VAR(t) \cap VAR(\vec{t'}) \neq \emptyset,\ VAR(t) \setminus VAR(\vec{t'}) = \{X_1, \ldots, X_l\}\} \cup\\
&\{abddom(X) :- nonRecAbddom(X)\},
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{guess}(V) = \{&a(X_1, \ldots, X_n) :- \text{not } \neg a(X_1, \ldots, X_n), abddom(X_1), \ldots, abddom(X_n),\\
&\neg a(X_1, \ldots, X_n) :- \text{not } a(X_1, \ldots, X_n), abddom(X_1), \ldots, abddom(X_n) \mid a/n \in \mathcal{P}_d\},
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{check}(I'_v) = \{&:- \text{not } v(t_1, \ldots, t_n) \mid v(t_1, \ldots, t_n) \in I'_v, v/n \in \mathcal{P}_i\} \cup\\
&\{vInVAS(t_1, \ldots, t_n) :- \mid v(t_1, \ldots, t_n) \in I'_v, v/n \in \mathcal{P}_i\} \cup\\
&\{:- v(X_1, \ldots, X_n), \text{not } vInVAS(X_1, \ldots, X_n) \mid v/n \in \mathcal{P}_i\}.
\end{aligned}
$$

Figure 3.9: Elements of the abduction program $\lambda(I'_v, V)$.

Therefore, not all visualisations can be successfully used for abduction. Consider the case of a visualisation program depending on absolute positions, mostly constructed of $visposition/4$ atoms. As $visposition/4$ is not contained in the set $\mathcal{P}_i$ of integrity predicates, it will not be used to build the check part of the abduction program and thus the inferred interpretation may not be useful. But for most visualisations, it is easy to find a visualisation of the problem which can be edited and afterwards I' can be inferred by using visualisation elements contained in $\mathcal{P}_i$. Just using relative positioning, grids and graphs, will help to create visualisations which can be edited successfully because they are contained in $\mathcal{P}_i$.

As described above, we have to determine the predicates and domains of the abducible atoms as well as choosing the set of integrity predicates from the set of visualisation predicates. To get the set $\mathcal{P}_a$ of predicates of abducible atoms, we extract them from the visualisation program. The idea behind this approach is that if the visualisation is complete, i.e., all elements are visualised by the user, most of the predicates of the solution of the domain program P should occur somewhere in the body of the visualisation program in order to visualise all elements of the solution. Thus, we get the set $\mathcal{P}_a$ by examining the body of every rule in the visualisation program V and collecting every predicate not contained in the set of integrity predicates.

After getting the abducible atoms, we then face the problem to determine the *abduction domain* $\mathcal{D}_a$, where we—as described above—use the visualisation program and construct rules to identify the constants and function symbols needed for the abduction domain. Consider the following rules:

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

where I_v consists of the following facts:

$$
\begin{aligned}
&visrect(f(bakerstreet, 221b), 9, 10),\\
&visellipse(sun, 10, 11).
\end{aligned}
$$

We do not need to include the function symbol $f/2$ in the domain, because it is only needed for the visualisation. On the other hand, if we only extract the constant as well as function symbols of I'_v, the function symbol $size/2$ would not be included in the abduction domain, which means that it would not be possible to infer a corresponding interpretation I'. The same problem holds also for deleted elements as well as for newly created elements in the modified visualisation. Thus, the rules of the abduction program, depicted in Figure 3.9, are used to undergo these problems.

In the above example, the following rules would be generated to retrieve the abduction domain:

$$\begin{aligned} abddom(Street) &:- visrectInVAS(f(Street, \_), \_, \_), \\ abddom(Num) &:- visrectInVAS(f(\_, Num), \_, \_), \\ abddom(size(Width, Height)) &:- visellipseInVAS(sun, Width, Height). \end{aligned}$$

Thus, here we would retrieve

$$\{bakerstreet, 221b, size(10, 11)\}$$

as the abduction domain. Note that we have to use fresh predicates which we construct by adding the postfix $inVAS$ to the specific visualisation predicates which are generated in the $\mathsf{check}(I'_v)$ part of Figure 3.9.

As stated above, concerning the integrity predicates $\mathcal{P}_i$, we take only those predicates into account which can be easily changed (i.e., enumerated) and which do not rely on preciseness. Thus, we exclude elements like $visposition/4$, because the user would need to position the elements such that this visualisation could be an answer set of the visualisation program. Otherwise, I''_v and I'_v would not coincide and thus the abduction program would not have any answer set. But, as said before, most problems can be represented very easily for editing purposes. Consider, for example, the Sudoku puzzle, where the task is to fill a grid with numbers such that certain rules are fulfilled. This could be either represented by straight lines defining the different cells where numbers should be put in or someone can define a grid where the numbers can be changed easily and later on also a corresponding interpretation I' can be inferred.

Next, let us have a look at the abduction program.

Definition 3.3. 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, the *abduction program* with respect to I'_v and V is given by

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

where $\mathsf{dom}(I'_v, V)$, $\mathsf{guess}(V)$, and $\mathsf{check}(I'_v)$ are given in Figure 3.9, for which

(i) $nonRecAbddom/1$, $abddom/1$, and $vInVAS/n$, for all $v/n \in \mathcal{P}_i$, are fresh predicates, where $vInVAS/n$ results from concatenation of the visualisation predicate v with the postfix $InVAS$,

(ii) $VAR(t)$ denotes the variables occurring in a term (or sequence of terms) t, and

(iii) $\mathcal{P}_d$ is given by

$$\mathcal{P}_d = \{a/n \mid \text{there are terms } t_1, \ldots, t_n \text{ such that } a(t_1, \ldots, t_n) \in \bigcup_{r \in V} \mathrm{B}(r) \text{ but there are no terms } t'_1, \ldots, t'_n \text{ such that } a(t'_1, \ldots, t'_n) \in \bigcup_{r \in V} \mathrm{H}(r)\} \setminus \mathcal{P}_v.$$

□

The rules in program $\mathsf{dom}(I'_v, V)$ are used to derive the abduction domain $\mathcal{D}_a$. The idea is that the body of the visualisation program mostly contains the non-visualisation predicates while the head mostly contains the visualisation predicates from $\mathcal{P}_v$ which describe the graphical representation. This information can be used to construct rules computing the abduction domain, where the visualisation predicates used in the body of the constructed rules may be in the head of a rule in the visualisation program and the non-visualisation predicate may only be in the body of the rules but not in the head of a rule in V. All non-visualisation predicates occurring in the head of a rule in V are auxiliary predicates and thus there may no domain be inferred for them. The predicate $nonRecAbddom/1$ is used to avoid infinite groundings.

The next part of the abduction program is $\mathsf{guess}(V)$, where the atoms of the domain program P are guessed, i.e., the abducible atoms. The output of the guessing part is I'. Finally, $\mathsf{check}(I'_v)$ contains all constraints and auxiliary facts. There are constraints which are responsible that all atoms contained in I'_v are also contained in the hypothetical visualisation interpretation I''_v, i.e., that no atom from I'_v is missing. Moreover, there are constraints which are responsible that the abduction derives not more or other visualisation atoms as contained in I'_v. Thus, if the abduction program successfully outputs an answer set, then I'_v coincides with I''_v and therefore intuitively $I' \cup V$ yields the modified visualisation interpretation I'_v.

At first sight, the computation of the abduction domain with the generated domain rules seems to be straightforward. However, it can happen that in the domain rules not all constants and function symbols needed for inferring I' are derived by the set of domain rules and consequently the abduction program does not yield an answer set in such a case. Consider the following rules:

$$viscolor(id1, red) :- color(id1, 1), m(1, red),$$
$$viscolor(id2, blue) :- color(id2, 2), m(2, blue).$$

The domain rules for this example are

$$nonRecAbddom(id1) :- viscolorInVAS(id1, red),$$
$$nonRecAbddom(red) :- viscolorInVAS(id1, red),$$
$$nonRecAbddom(id2) :- viscolorInVAS(id2, blue),$$
$$nonRecAbddom(blue) :- viscolorInVAS(id2, blue),$$
$$abddom(X) :- nonRecAbddom(X).$$

Here, the auxiliary predicate $m/2$ is used mapping the numbers assigned in the domain program to colours. Since the numbers are not used in the visualisation, they cannot be calculated for the abduction domain, and thus there cannot be any answer set of the abduction program because the mapping is fixed but the numbers are missing in the abduction domain.

For a situation as in the above example, i.e., where the abduction program has no answer set, Kara allows the user to edit the abduction domain before the abduction program is constructed and executed.

The following two results characterise the answer sets of the abduction program.

Theorem 3.4. *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 $\mathcal{P}_i$.*

Proof. Towards a contradiction, assume that I''_v is an answer set of $\lambda(I'_v, V)$ such that $M_{I'_v} \neq M_{I''_v}$, where

$$M_{I'_v} = \{p(t_1, t_2, \ldots, t_n) \in I'_v \mid p/n \in \mathcal{P}_i\} \quad \text{and}$$
$$M_{I''_v} = \{p(t_1, t_2, \ldots, t_n) \in I''_v \mid p/n \in \mathcal{P}_i\}.$$

We have to check two cases, namely where either there is some $p(t_1, t_2, \ldots, t_n) \in M_{I'_v} \setminus M_{I''_v}$ or there is some $p(t_1, t_2, \ldots, t_n) \in M_{I''_v} \setminus M_{I'_v}$.

Assume there is some $p(t_1, t_2, \ldots, t_n) \in I'_v$ such that $p(t_1, t_2, \ldots, t_n) \notin I''_v$, where $p \in \mathcal{P}_i$. As $p/n \in \mathcal{P}_i$, it follows that

$$:- \text{ not } p(t_1, t_2, \ldots, t_n) \in \mathsf{check}(I'_v).$$

Hence, $p(t_1, t_2, \ldots, t_n) \in I''_v$ because otherwise I''_v would not be an answer set of the abduction program. But this violates our assumption $p(t_1, t_2, \ldots, t_n) \notin I''_v$. Hence, $M_{I'_v} \setminus M_{I''_v} = \emptyset$, that is, $M_{I'_v} \subseteq M_{I''_v}$.

Now assume that $p(t_1, t_2, \ldots, t_n) \in I''_v$ and $p(t_1, t_2, \ldots, t_n) \notin I'_v$, for some $p \in \mathcal{P}_i$. By the definition of $\mathsf{check}(I'_v)$, we have

$$:- p(t_1, t_2, \ldots, t_n), \text{not } pInVAS(t_1, t_2, \ldots, t_n) \in \mathsf{check}(I'_v),$$

where $pInVAS$ is a fresh predicate and results from the concatenation of the predicate p with the postfix $InVAS$.

Since $p(t_1, t_2, \ldots, t_n) \in I''_v$, it follows that $pInVAS(t_1, t_2, \ldots, t_n) \in I''_v$ must hold. Since $pInVAS$ is a fresh predicate, we get that $p(t_1, t_2, \ldots, t_n) \in I'_v$. This contradicts our assumption that $p(t_1, t_2, \ldots, t_n) \notin I'_v$. Hence, $M_{I''_v} \setminus M_{I'_v} = \emptyset$, and so $M_{I''_v} \subseteq M_{I'_v}$. Consequently, we obtain that $M_{I'_v} = M_{I''_v}$. □

Theorem 3.5. *Let I'_v be an interpretation with atoms over $\mathcal{P}_v$, $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates, and V a (visualisation) program such that for every I' with atoms over $\mathcal{P}_d$, where*

$$\mathcal{P}_d = \{a/n \mid \textit{there are terms } t_1, \ldots, t_n \textit{ such that } a(t_1, \ldots, t_n) \in \bigcup_{r \in V} \mathrm{B}(r) \textit{ but there are no terms } t'_1, \ldots, t'_n \textit{ such that } a(t'_1, \ldots, t'_n) \in \bigcup_{r \in V} \mathrm{H}(r)\} \setminus \mathcal{P}_v,$$

every two answer sets I_1 and I_2 of $V \cup I'$ do not differ on $\mathcal{P}_i$. Then, for any answer set I''_v of $\lambda(I'_v, V)$, a solution I' of the abduction problem $(*)$ is obtained by projecting I''_v to the predicates in $\mathcal{P}_d$.

Proof. Consider some $I''_v \in AS(\lambda(I'_v, V))$ and let I' be obtained by projecting I''_v to the predicates in $\mathcal{P}_d$, i.e., $I' = \{p(t_1, t_2, \ldots, t_n) \in I''_v \mid p/n \in \mathcal{P}_d\}$. As I' contains only atoms already present in I''_v, we have that $I''_v \in AS(\lambda(I'_v, V) \cup I')$. Furthermore, we can remove the guess part from the abduction program because these rules construct exactly those literals which are contained in I' and obtain that $I''_v \in AS((\lambda(I'_v, V) \setminus \mathsf{guess}(V)) \cup I')$. We can moreover remove $\mathsf{dom}(I'_v, V)$ from $\lambda(I'_v, V) \setminus \mathsf{guess}(V)$ because this part is only needed for the guess part of $\lambda(I'_v, V)$. Hence, $I'''_v \in AS((\lambda(I'_v, V) \setminus (\mathsf{guess}(V) \cup \mathsf{dom}(I'_v, V))) \cup I')$, where I'''_v is the projection of I''_v to the atoms with predicate symbols $dom/1$ or $nonRecDom/1$. Lastly, we can remove $\mathsf{check}(I'_v)$ from the abduction program because it only adds fresh predicates $vInVAS/n$ to I''_v. Hence, we have $I^{iv}_v \in AS((\lambda(I'_v, V) \setminus (\mathsf{guess}(V) \cup \mathsf{dom}(I'_v, V) \cup \mathsf{check}(I'_v))) \cup I')$, for

$$I^{iv}_v = \{p(t_1, t_2, \ldots, t_n) \in I'''_v \mid p \neq vInVAS \text{ for every } v \in \mathcal{P}_i\}.$$

Since $\lambda(I'_v, V) \setminus (\mathsf{guess}(V) \cup \mathsf{dom}(I'_v, V) \cup \mathsf{check}(I'_v)) = V \cup I'$, we have that $I^{iv}_v \in AS(V \cup I')$. As I^{iv}_v coincides with I''_v on the atoms with predicates from $\mathcal{P}_i$, and since by Theorem 3.4 we have that I''_v coincides with I'_v on the atoms with the predicates from $\mathcal{P}_i$, also I^{iv}_v coincides with I'_v on the atoms with the predicates from $\mathcal{P}_i$. Moreover, as $I^{iv}_v \in AS(V \cup I')$ and every two answer sets I_1 and I_2 of $V \cup I'$ do not differ on $\mathcal{P}_i$, I'_v coincides with each answer set of $V \cup I'$ on the atoms with predicates from $\mathcal{P}_i$. □

Abduction example

We illustrate the generation of the abduction program using the *maze generation problem* [33]. A maze is a two-dimensional grid where every cell is either a wall or is empty and there is exactly one entrance and one exit in the maze. The maze generation problem is the task to construct mazes given an initial state such that the following conditions are met:

1. Each cell in an edge of the grid is a wall, except entrance and exit that are empty.
2. There is no 2×2 square of empty cells or walls.
3. If two walls are on a diagonal of a 2×2 square, then not both of their common neighbors are empty.
4. No wall is completely surrounded by empty cells.
5. There is a path from the entrance to every empty cell (a path is a finite sequence of cells, in which each cell is horizontally or vertically adjacent to the next cell in the sequence).

The maze generation problem was a benchmark for the Second and Third ASP Competition [23, 34]. An encoding of the maze generation problem is given in Figure 5.2, which is taken from the Third ASP Competition's web page. The output of the maze generation problem contains the predicates $wall/2$ and $empty/2$ which must be derived by the domain program P in order to find correct mazes.

$$
\begin{aligned}
grid(X,Y) :- \ & col(X), row(Y).\\
adjacent(X,Y,X,Y1) :- \ & grid(X,Y), Y1 = Y+1, row(Y1).\\
adjacent(X,Y,X,Y1) :- \ & grid(X,Y), Y1 = Y-1, row(Y1).\\
adjacent(X,Y,X1,Y) :- \ & grid(X,Y), X1 = X+1, col(X1).\\
adjacent(X,Y,X1,Y) :- \ & grid(X,Y), X1 = X-1, col(X1).\\
\\
border(1,Y) :- \ & row(Y).\\
border(X,1) :- \ & col(X).\\
border(X,Y) :- \ & row(Y), maxCol(X).\\
border(X,Y) :- \ & col(X), maxRow(Y).\\
\\
empty(X,Y) :- \ & input_empty(X,Y).\\
wall(X,Y) :- \ & input_wall(X,Y).\\
\\
wall(X,Y) \vee empty(X,Y) :- \ & grid(X,Y), \text{not } border(X,Y), \text{not } entrance(X,Y),\\
& \text{not } exit(X,Y).\\
\\
wall(X,Y) :- \ & border(X,Y), \text{not } entrance(X,Y), \text{not } exit(X,Y).\\
\\
empty(X,Y) :- \ & entrance(X,Y).\\
empty(X,Y) :- \ & exit(X,Y).\\
\\
:- \ & wall(X,Y), wall(X1,Y), wall(X,Y1), wall(X1,Y1),\\
& X1 = X+1, Y1 = Y+1.\\
:- \ & empty(X,Y), empty(X1,Y), empty(X,Y1),\\
& empty(X1,Y1), X1 = X+1, Y1 = Y+1.\\
\\
:- \ & wall(X,Y), wall(Xp1,\ Yp1), empty(Xp1,Y),\\
& empty(X,\ Yp1), Xp1 = X+1,\ Yp1 = Y+1.\\
:- \ & wall(Xp1,Y), wall(X,\ Yp1), empty(X,Y),\\
& empty(Xp1,\ Yp1), Xp1 = X+1,\ Yp1 = Y+1.\\
\\
:- \ & wall(X,Y), \text{not } border(X,Y),\\
& \text{not } wallWithAdjacentWall(X,Y).\\
wallWithAdjacentWall(X,Y) :- \ & wall(X,Y), adjacent(X,Y,W,Z), wall(W,Z).\\
\\
reach(X,Y) :- \ & entrance(X,Y).\\
reach(XX,\ YY) :- \ & adjacent(X,Y,XX,\ YY), reach(X,Y), empty(XX,\ YY).\\
:- \ & empty(X,Y), \text{not} reach(X,Y).
\end{aligned}
$$

Figure 3.10: An encoding of the maze generation problem.

The following rules constitute a simple visualisation program, taking for the sake of simplicity only the $wall/2$ predicates into account:

$$visrect(f(X,Y),20,20) :- \ wall(X,Y). \tag{3.61}$$

$$visbackgroundcolor(f(X,Y),black) :- \ wall(X,Y). \tag{3.62}$$

$$visposition(f(X,Y),X*20,Y*20,0) :- \ wall(X,Y). \tag{3.63}$$

$$visdeletable(ID) :- \ visrect(ID,\_,\_). \tag{3.64}$$

Figure 3.11: A possible visualisation of the output of the maze generation problem.

Intuitively, in Rule (3.61), we generate for every $wall/2$ predicate a rectangle with the newly constructed identifier $f(X, Y)$ and a width and height of 20 pixels. Rule (3.62) sets for every rectangle constructed from a $wall/2$ predicate the background colour to black and Rule (3.63) sets the absolute position of every rectangle on the grid. The position is calculated by the (x, y)-coordinates of the wall on the grid and the width and height specified in the visualisation program. Rule (3.64) defines every created rectangle of the visualisation as deletable, enabling that the visualisation can be edited. The corresponding visualisation can be found in Figure 3.11.

An extract of the corresponding visualisation answer set looks as follows:

$$
\begin{aligned}
&visbackgroundcolor(f(1,1), black),\ visbackgroundcolor(f(1,3), black), \ldots,\\
&visdeletable(f(1,1)),\ visdeletable(f(1,3)) \ldots\\
&visposition(f(1,1), 20, 20, 0),\ visposition(f(1,3), 20, 60, 0), \ldots,\\
&visrect(f(1,1), 20, 20), visrect(f(1,3), 20, 20), \ldots.
\end{aligned}
$$

Let us now edit the visualisation by deleting one or more walls in the example, e.g., the wall at position $(3, 4)$, and we want to infer the corresponding interpretation I'. If inferred, it will consist only of atoms with predicate $wall/2$, because that was the only predicate used in our visualisation program.

First, we create the domain rules:

$$
\begin{aligned}
nonRecAbddom(X) &:- visrectInVAS(f(X,Y), 20, 20),\\
nonRecAbddom(Y) &:- visrectInVAS(f(X,Y), 20, 20),\\
nonRecAbddom(X) &:- visbackgroundcolorInVAS(f(X,Y), black),\\
nonRecAbddom(Y) &:- visbackgroundcolorInVAS(f(X,Y), black),\\
nonRecAbddom(X) &:- vispositionInVAS(f(X,Y), X*20, Y*20, 0),\\
nonRecAbddom(Y) &:- vispositionInVAS(f(X,Y), X*20, Y*20, 0),\\
abddom(X) &:- nonRecAbddom(X).
\end{aligned}
$$

Recall that we need fresh predicates (with postfix $InVAS$) for every used visualisation predicate. As can be seen, for every occurrence of $wall/2$ in the body of the visualisation program, domain rules are generated, and there is exactly one domain rule for every term used in the $wall/2$ predicate.

The next step is to define the guessing rules. As abducible atoms we have only those containing the $wall/2$ predicate, and we get the following two guessing rules for the walls of the grid:

$$
\begin{aligned}
wall(A,B) &:- abddom(A), abddom(B), \text{not } -wall(A,B),\\
-wall(A,B) &:- abddom(A), abddom(B), \text{not } wall(A,B).
\end{aligned}
$$

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg height="103" width="103" xmlns="http://www.w3.org/2000/svg">
  <rect fill="black" height="20" stroke="black" width="20" x="1" y="41"/>
  <rect fill="black" height="20" stroke="black" width="20" x="1" y="81"/>
  <rect fill="black" height="20" stroke="black" width="20" x="41" y="81"/>
  <rect fill="black" height="20" stroke="black" width="20" x="1" y="1"/>
  <rect fill="black" height="20" stroke="black" width="20" x="61" y="81"/>
  <rect fill="black" height="20" stroke="black" width="20" x="41" y="1"/>
  <rect fill="black" height="20" stroke="black" width="20" x="81" y="61"/>
  <rect fill="black" height="20" stroke="black" width="20" x="81" y="41"/>
  <rect fill="black" height="20" stroke="black" width="20" x="81" y="1"/>
  <rect fill="black" height="20" stroke="black" width="20" x="41" y="61"/>
  <rect fill="black" height="20" stroke="black" width="20" x="61" y="1"/>
  <rect fill="black" height="20" stroke="black" width="20" x="1" y="61"/>
  <rect fill="black" height="20" stroke="black" width="20" x="21" y="1"/>
  <rect fill="black" height="20" stroke="black" width="20" x="21" y="81"/>
  <rect fill="black" height="20" stroke="black" width="20" x="41" y="21"/>
  <rect fill="black" height="20" stroke="black" width="20" x="81" y="81"/>
</svg>
```

Figure 3.12: An SVG file for the maze generation example.

Concerning the check part of the abduction program, for this, we first turn atoms with predicates from $\mathcal{P}_i$ contained in the visualisation answer set into constraints. In our example, this means that we have no constraints for $visdeletable/1$ and $visposition/4$ as they are not contained in $\mathcal{P}_i$. Here is a short extract of three constraints from the first part of the check program:

$$\begin{aligned} &:- \text{not } visbackgroundcolor(f(1,1), black),\\ &:- \text{not } visbackgroundcolor(f(1,3), black),\\ &:- \text{not } visrect(f(1,1), 20, 20). \end{aligned}$$

Afterwards, we need to define the facts for the fresh predicates and the remaining constraints involving predicates from $\mathcal{P}_v$ and the fresh ones. Again, we only take the predicates contained in $\mathcal{P}_i$ into account. A short extract for the $visrect/3$ predicate is as follows:

$$\begin{aligned} &visrectInVAS(f(1,1), 20, 20),\\ &visrectInVAS(f(1,3), 20, 20),\\ &\qquad\vdots\\ &:- visrect(A, B, C), \text{not } visrectInVAS(A, B, C). \end{aligned}$$

Finally, we add the visualisation program to the abduction program. After executing this abduction program with a solver, we get all walls like before except for the removed one, and thus we have successfully inferred the corresponding interpretation I'.

Export

Kara supports exporting the graphical visualisation in the SVG format defined by the W3C [35]. The information about the rendering of the components is given by the so-called $EditPart$s of

the GEF. Every component respectively element has a corresponding $EditPart$ in the implementation and thus for every $EditPart$ instance one element in the SVG is generated. An SVG is a pure XML file and is therefore freely scalable and every tag defines an element with its properties. Thus, an $EditPart$ in the `Java` implementation corresponds to one XML tag in the SVG file and the properties can be set accordingly. The example of the maze generation problem converted to an SVG file is depicted in Figure 3.12. Note that if the user applies images to visualisations, like *gif* or *png* files, these cannot be directly converted to the SVG. Instead, they are converted to the ***Base64*** format by the internal `Java` implementation and then in their encoded string representation written to the SVG file.

CHAPTER 4

Applying Kara

This chapter explains the systems features from the users view and how the Kara plug-in can be used.

4.1 Getting started

When Eclipse is started, the first step is to install the SeaLion as well as the Kara plug-in. At present, SeaLion and Kara are contained in a single package which can be installed using the following URL:

http://mmdasp.sourceforge.net/sealion/update.

After the whole plug-in is installed, the answer-set programming IDE can be used including all basic features as well as the visualisation component.

First, a new project should be created as shown in Figure 4.1. Afterwards, it can be opened by right clicking the project and clicking on "Open Project". A new answer-set program can be written by creating a file by right clicking on the project and selecting "New" and "File". A screenshot showing part of the editor for answer-set programs is depicted in Figure 4.2, where on the left-hand side also the project listing is given. All projects are currently open and the files contained in the project "Test" can be seen, namely domainprog.gr and vis.gr. On the right, the answer-set programming editor opened with file vis.gr can be seen. If the cursor is moved over a predicate, then all occurrences of that predicate are highlighted with green background colour whereas constants and variables are highlighted with a thin black border.

4.2 Running a solver

The plug-in is highly configurable and allows to add as many external tools as the user wants. An external tool can be, for instance, a grounder or a solver which is needed to execute the answer-set programs. When a new external tool is added to the plug-in configuration, the user can

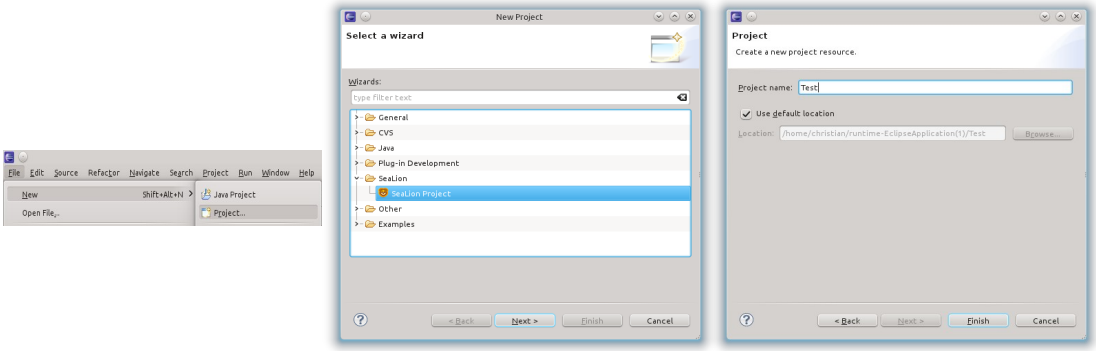

Figure 4.1: The steps how to create a new project in Eclipse.

also set default options which should normally be set when invoking the grounder respectively solver. The external tool configuration dialog can be accessed via the file menu "Window" – "Preferences" which contains an entry on the left called "SeaLion". Afterwards, on the right-hand side of the dialog, the external tool configuration setup like shown in Figure 4.3 can be seen. The figure shows that Gringo, Clasp, as well as DLV are installed, which are also supported in parsing their answer sets whereas the output of other solvers may not be able to be parsed by SeaLion. Note that it is nevertheless possible to execute *any* solver and read its output on the console embedded in Eclipse.

Run configurations

In Eclipse, there is a feature called *run configurations* where one can make configurations how a program should be executed. As Eclipse is implemented as a plug-in system, SeaLion uses this feature to execute the answer-set programs written by the user with a specific grounder and solver. The user creates new run configurations, where several settings can be changed. For instance, the programs to be executed can be defined as well as the preferred solver for the execution (e.g., piped tool or DLV) and also the command line parameters for the solver. After the parameters of the run configuration are defined, it can be executed by the user and saved with a name. Thus, for subsequent executions, only a single click on the run configuration is required. Figure 4.4 shows how the run configurations dialog can be accessed and how it looks like.

Furthermore, the processing strategy of answer sets can also be chosen if the user wants to parse it. Currently, there are three processing strategies implemented in the SeaLion core, namely

- DLV,
- Clasp, and

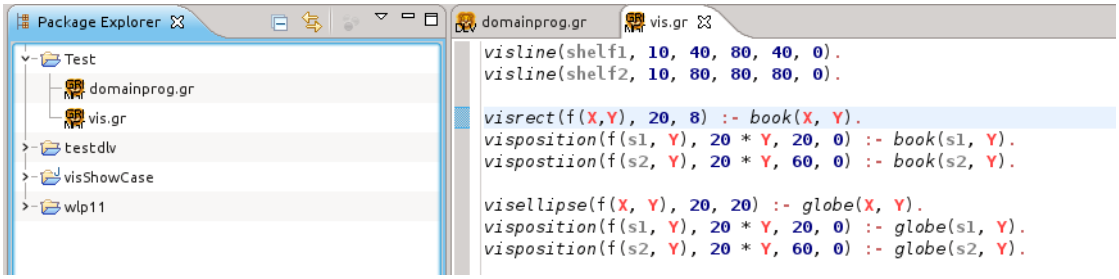

Figure 4.2: On the left-hand side: the project list; on the right-hand side: the editor with semantic highlighting.

- `ASP09`.[1]

Interpretation view

If the user chooses no answer-set processing strategy, the output is directly forwarded to the embedded console in `Eclipse`. Otherwise, if a processing strategy is chosen, the *interpretation view* is used to output the answer set(s). This is a graphical tree-like visualisation of answer sets where at the top (as the root node of the tree) the name of the answer set is given. On the second level, the names of the predicates contained in the answer set are provided, while the third level lists the constant and variable symbols of the predicates in the answer set.

For illustration, consider a simple domain program comprising the following rules:

$$book(s1, 1), \tag{4.1}$$
$$book(s1, 3), \tag{4.2}$$
$$book(s2, 1), \tag{4.3}$$
$$globe(s2, 2). \tag{4.4}$$

When executed with a solver like `Gringo`, and if the answer set is parsed by `SeaLion`, we get the result shown in Figure 4.5.

So far, we described how to execute answer-set programs inside the `SeaLion` system. Next, we describe how to visualise them using `Kara`.

4.3 Visualisation

For visualising answer sets, we need a domain program as well as a visualisation program, where the latter is responsible for realising the graphical representation. Most often the user wants to get the answer set for visualisation from a domain program which is executed by a solver as described above.

[1] `ASP09` was the output format chosen for the Second ASP Competition [23].

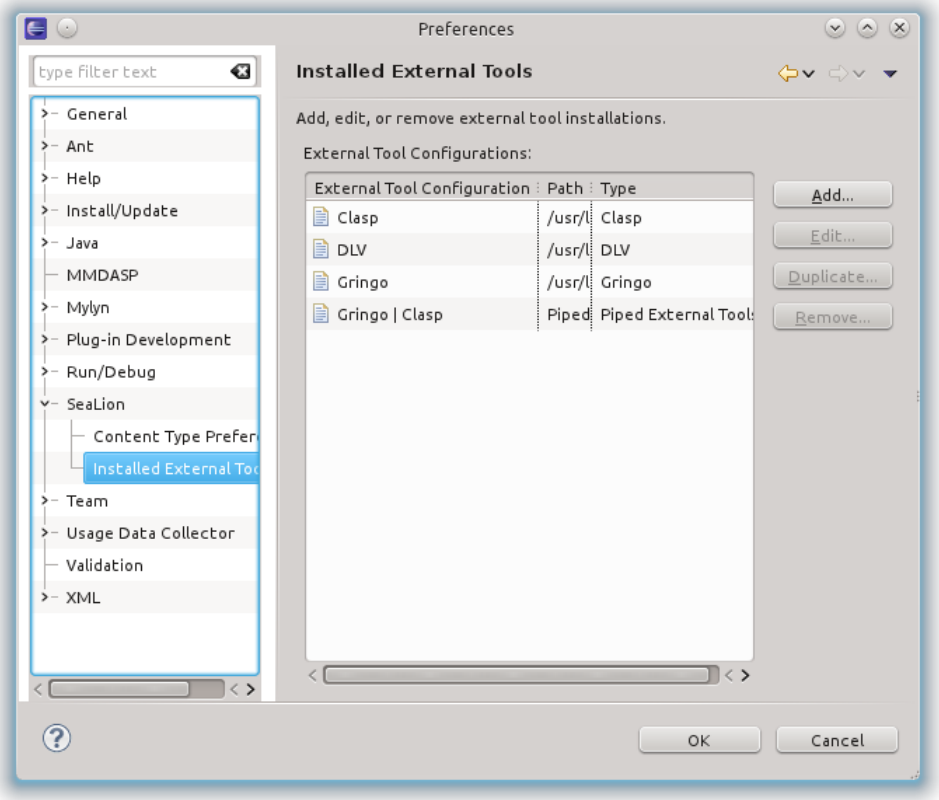

Figure 4.3: External tools configuration dialog.

Let us take as example the answer set shown in Figure 4.5. Given this answer set, we need to define a mapping from this answer set to a visualisation. This can be realised by a visualisation program comprising the following rules:

$$visline(shelf1, 10, 40, 80, 40, 0), \tag{4.5}$$
$$visline(shelf2, 10, 80, 80, 80, 0), \tag{4.6}$$
$$visrect(f(X,Y), 20, 8) :- \ book(X,Y), \tag{4.7}$$
$$visposition(f(s1,Y), 20 * Y, 20, 0) :- \ book(s1,Y), \tag{4.8}$$
$$visposition(f(s2,Y), 20 * Y, 60, 0) :- \ book(s2,Y), \tag{4.9}$$
$$visellipse(f(X,Y), 20, 20) :- \ globe(X,Y), \tag{4.10}$$
$$visposition(f(s1,Y), 20 * Y, 20, 0) :- \ globe(s1,Y), \tag{4.11}$$
$$visposition(f(s2,Y), 20 * Y, 60, 0) :- \ globe(s2,Y). \tag{4.12}$$

This answer-set program defines the mapping between the answer set from the interpretation view and the graphical representation. Rules (4.5) and (4.6) take care for drawing for each shelf

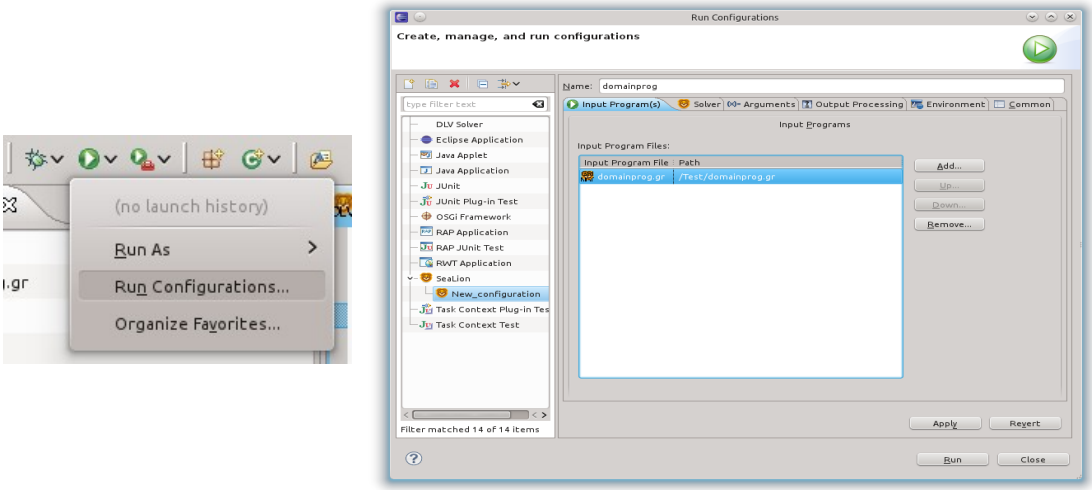

Figure 4.4: Left: access of the run configurations; right: the run configurations dialog.

a line which represent the shelf in the graphical representation. The constants $shelf1$ and $shelf2$ are the identifiers of each shelf which should be unique in the visualisation program to guarantee that the visualisation is rendered as intended. The first two arguments after the identifier define the starting position of the line in form of (x, y)-coordinates. The next two arguments define the end position of the line and the last argument is the z-coordinate which defines the visible element if two of them are overlapping.

Rule (4.7) defines that every book is rendered as a rectangle. To this end, we construct a new identifier from the shelf and row position of the book by using the function symbol f. The first parameter after the identifier defines the height of the rectangle, whereas the last parameter defines the width of it. Rules (4.8) and (4.9) are used to define the absolute position of the book on the editor pane. For the x-position of the book, we multiply the logical row with 20 because every component has a width of 20, and for the y-position, we can take a fixed value because each shelf has a fixed position on the editor pane. The z-coordinate is not relevant in this example and thus we set it to 0.

We could use the same code used for the books also for drawing globes, but we want to distinguish between these two. Thus, we represent globes as circles by using a diameter of 20 which is defined by Rule (4.10). The positioning of globes is exactly the same as for rectangles, which is realised by Rules (4.11) and (4.12).

After we have written the visualisation program, we can execute the visualisation by first selecting an answer set from the interpretation view. An interpretation can be selected by right clicking on it and then clicking on the “Visualisation” entry as seen in Figure 4.6. Note that by clicking on “Automatic visualisation” no visualisation program is needed and only a graphical representation which represents the source code of the answer-set program is rendered.

After clicking on visualisation, the run configurations dialog for visualisations open which

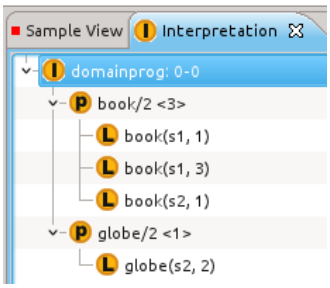

Figure 4.5: Sample output in the *interpretation view* of SeaLion.

extends the run configurations dialog of the SeaLion core. On the first tab, called "Visualisation input", the user can define the visualisation file and the interpretation which should be used for visualisation, where the selected interpretation from the interpretation view is preselected. The next tab, "Solver", is the same as in the SeaLion core and is used to select the solver with which the user wants to execute the visualisation. The only new tab is the tag for the "Positioning script", which defines the script which should be used for relative positioning of elements if the user needs it. Currently, versions for Clasp and DLV are provided. After all settings are chosen by the user, the visualisation can be executed. In our example, we get the result depicted in Figure 3.4.

Kara offers a wide variety of visualisation predicates and possibilities. A list with a description for each visualisation element is given in Table 3.1. Examples for the usage of these visualisation predicates is given in Chapter 5.

In addition to the predefined visualisation predicates, there are also some predefined constants in Kara for some predicates, which are given in Table 4.1. Additionally, Kara supports also numbers for $color/2$ and $backgroundcolor/2$. If numbers are used, every number is mapped internally to a colour by Kara and the number is replaced by the corresponding colour. Moreover, there are further constants for defining changeable properties; the full listing is given in Table 4.2.

Visualising graph structures

Graph structures can easily be created and visualised in Kara, because the graph needs only be defined and Kara does the positioning of the graph for the user. The nodes and connections of a graph can be rendered individually as the user wishes to. For instance, the user can render the nodes of a graph as rectangles, ellipses, polygons, etc. Graphical elements are defined as usual, but additionally they are also defined as *nodes*, which means that the elements are prepared to take part in a graph. The graph itself is defined via the predicate symbol $graph/1$ and the node predicate needs as arguments the identifier of the element, which should be a node, as well as the identifier of the graph in which the node should be part of.

Example 4.1. The following program renders two nodes of a graph as rectangles with a single

$backgroundcolor/2$	blue, white, green, red, darkblue, darkgray, gray, cyan, orange, lightblue, lightgray, lightgreen, yellow, black
$color/2$	blue, white, green, red, darkblue, darkgray, gray, cyan, orange, lightblue, lightgray, lightgreen, yellow, black
$vistargetdeco/2$	none, arrow, arrowfilled
$vissourcedeco/2$	none, arrow, arrowfilled
$visfontstyle/2$	bold, underline, italic

Table 4.1: Predefined constants for some predicates.

connection between them:

$$\begin{aligned} \{graph(g) :- \ , \\ rect(a) :- \ , \\ rect(b) :- \ , \\ node(a, g) :- \ , \\ node(b, g) :- \ , \\ connect(c, a, b) :- \ \}. \end{aligned}$$

The graph is positioned automatically by `Kara`. ♢

Visualising grids

Similar to rendering graph structures, for visualising grids, we define the graphical elements as usual, using a special predicate $visfillgrid/4$ to insert the elements into the grid. The grid itself is defined with predicate $visgrid/5$, where arguments define the identifier, row count, column count, height, as well as the width of the whole grid. The predicate $visfillgrid/4$ takes the identifier of the grid and the element as argument, as well as the row and the column where to insert the element in the grid.

Example 4.2. Let g be a grid and r a rectangle. Then, the insert in column 3 and row 3 can be achieved by $visfillgrid(g, r, 3, 3)$. ♢

4.4 Editing

We now describe how the editing features of `Kara` can be used. First, the editing possibilities of a visualisation depend on the visualisation predicates defined in the visualisation program. The three types of editing features supported are

- the *change of properties* of graphical elements,

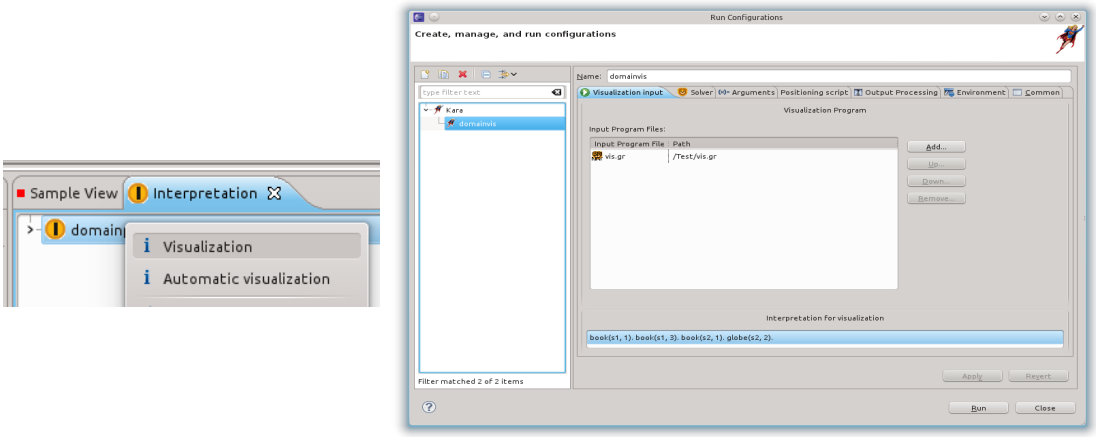

Figure 4.6: Steps to execute the visualisation.

- the *deletion* of elements, and
- the *creation* of elements.

Changing properties

For changing properties, the user marks the properties which he or she wants to be changeable with the visualisation predicate $vischangable/2$. The first argument is the identifier of the corresponding element and the second argument is the property which should be changeable. Table 4.2 shows which properties may be changeable if defined by the user for each graphical element. As well, Table 4.3 lists constants which may be defined to be changeable.

If the user defines for a graphical element that some property is changeable, it can be changed in the visualisation accordingly.

Example 4.3. The following fact defines that the background colour for some graphical element with identifier $maximus$ is changeable:

$$vischangable(maximus, backgroundcolor).$$

In the visualisation itself, the properties of the element can be changed by right clicking on the specific element and selecting “Edit properties”. ♢

Changing elements in a grid

A grid can also be changed respectively edited, and thus we define so-called *changeable elements* for a grid. This is realised in terms of the predicate $vispossiblegridvalues/2$ which takes as arguments the identifier of the grid and the identifier of an element which can be inserted into the grid.

connection	foreground colour, line width, line style, label, source decoration, target decoration
ellipse	foreground colour, line width, line style, label, background colour, connection, height, width
image	height, width, connection
labels	foreground colour, background colour, connection, font family, font size, font style, text
line	foreground colour, line width, line style, label
polygon	foreground colour, background colour, connection, label, line style, line width
rectangle	foreground colour, background colour, connection, height, width, line style, line width

Table 4.2: Properties which can be changed for each element.

Example 4.4. Let g be a grid, b a rectangle with black background colour, and y a rectangle with yellow background colour. Then,

$$vispossiblegridvalues(g, b),$$
$$vispossiblegridvalues(g, y)$$

defines that at every position in the grid, a black or a yellow rectangle can be inserted instead of the element which is currently available in this cell. ♢

In the user interface, the element of each grid cell can be changed by right clicking on the corresponding element and then clicking on "Change element". A dialog opens where the new element can be chosen for insertion.

Deleting elements

If defined in the visualisation program, elements can also be deleted which is done with the predicate $visdeletable/1$, taking only the identifier of the deletable element as argument. These elements can be deleted in the user interface by right clicking them and choosing "Delete".

Creating elements

Creatable elements can be defined with the predicate $viscreatable/1$. This predicate takes only the identifier of the element as argument which should be creatable. All elements which are creatable appear on the right-hand side in the "Palette" of the GEF graphical editor with their identifier as description. They can be created by simply dragging them into the editor pane. `Kara` then generates a new identifier for them.

connection	connection
backgroundcolor	background colour
color	foreground colour
linewidth	line width
fontfamily	font family
text	text
linestyle	Line style
height	height
width	width
fontstyle	font style
fontsize	font size
sourcedeco	source decoration
targetdeco	target decoration
label	label
color	foreground colour

Table 4.3: Changeable constants.

Changing the identifier of an element

It is also possible to change the identifier of a single object if the user wants or needs to do it. This can be achieved by right clicking the corresponding element and choosing "Change identifier" and entering a new one.

4.5 Export

Visualisations made with `Kara` can also be exported in SVG format to achieve better portability of these visualisations. There are two types of visualisations, namely export the whole visualisation or export only selected parts of the visualisation.[2] After clicking the export menu entry, the user can choose the path where the exported visualisation should be stored in the file system. If images are used in the visualisation, they are encoded in SVG using `Base64` encoding such that no references to external images are needed.

4.6 Inferring an interpretation

If we already have a visualisation and possibly changed it, a visualisation interpretation can be obtained from the graphical editor. This is achieved by right clicking somewhere in the editor and executing the action "Show interpretation". Afterwards, the interpretation corresponding to the visualisation appears in the interpretation view of `SeaLion`. However, this interpretation contains only visualisation predicates and maybe also auxiliary predicates necessary for the

[2] They may be selected using the selection tool.

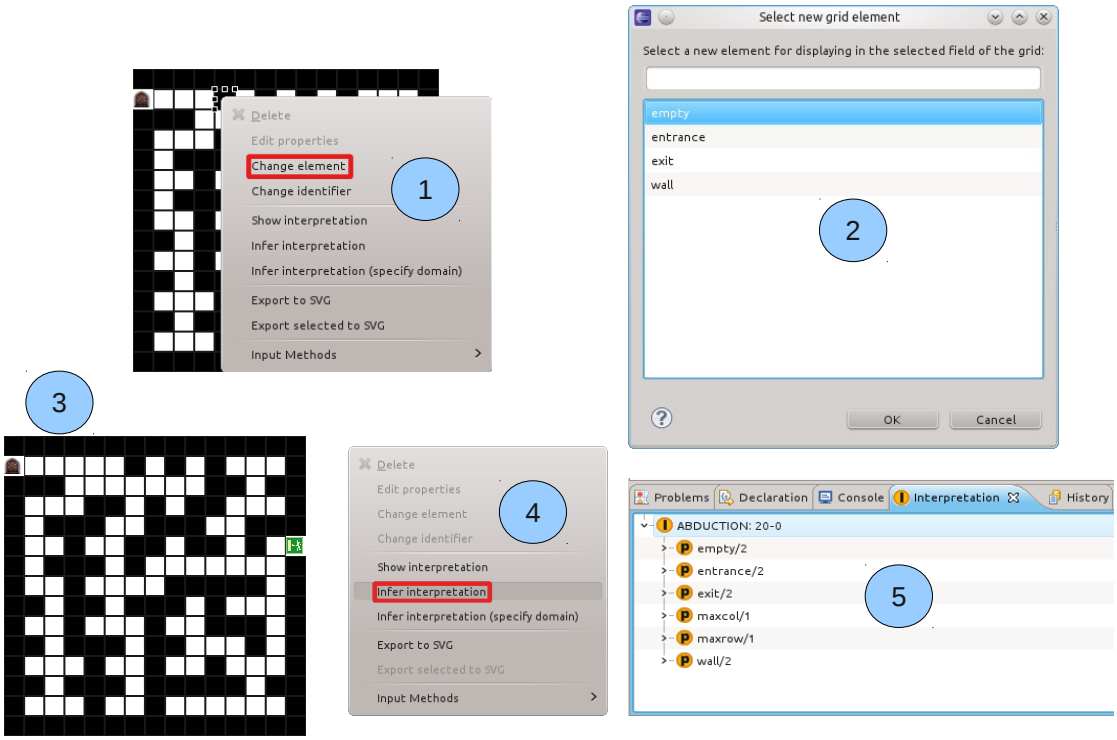

Figure 4.7: Abduction steps in `Kara`.

visualisation program. Now, *prima facie* we are not interested in the visualisation interpretation but in the corresponding interpretation of the domain program. This can be done by *inferring this interpretation*, by simply choosing the "Infer interpretation" action of the context menu. Then, the corresponding interpretation with the domain predicates is calculated. If there is a solution, i.e., an answer set to the abduction problem, the corresponding interpretation appears in the interpretation view under the name "ABDUCTION".

There is also a second possibility for inferring an interpretation. To wit, the user can specify the abduction domain by himself or herself. This can be useful if the user, e.g., creates some new elements and the abduction domain cannot be exactly calculated by `Kara`. It is possible to list several constants as strings, but it is also possible to specify number ranges (e.g., 0-100).

A summary of the abduction steps are depicted in Figure 4.7.

CHAPTER 5

Examples

This chapter demonstrates the features of `Kara` by showing various examples of visualisation programs.

5.1 Basic example

The first example is a standalone visualisation dealing with the concept of relative positioning. We want to draw a house with a roof and on top of the house we want to draw a sun on the left side. Here, we do not need any domain program because we write directly the visualisation literals as facts into the visualisation program. However, note that we have to join the visualisation program with an empty interpretation in order to execute it with the `Kara` system.

The program consists of the following facts:

$$visrect(house, 20, 20), \tag{5.1}$$
$$visbackgroundcolor(house, red), \tag{5.2}$$
$$visellipse(sun, 15, 15), \tag{5.3}$$
$$visbackgroundcolor(sun, yellow), \tag{5.4}$$
$$vispolygon(roof, 0, 20, 1), \tag{5.5}$$
$$vispolygon(roof, 10, 0, 2), \tag{5.6}$$
$$vispolygon(roof, 20, 20, 3), \tag{5.7}$$
$$visabove(roof, house), \tag{5.8}$$
$$visabove(sun, house), \tag{5.9}$$
$$visabove(sun, roof), \tag{5.10}$$
$$visleft(sun, house), \tag{5.11}$$
$$visleft(sun, roof). \tag{5.12}$$

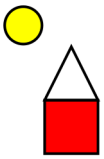

Figure 5.1: Visualisation of the house example using relative positioning.

Fact (5.1) creates a new rectangle, having identifier $house$ and a width and height of 20 pixels. Fact (5.2) uses the identifier $house$ of the rectangle to set its background colour to red. `Kara` offers a set of predefined colours which the user can apply as foreground and background colours to available elements. Other colours than these predefined ones cannot be used. Fact (5.3) creates a new ellipse (actually, in our case, a circle) with a width and height of 15 pixels and, analogously to the above, on the rectangle the background colour is set to yellow by referring to the element with its identifier. Then we need to create an roof, which we want to draw as a triangle which is not offered as a "ready to use" element by `Kara`. Thus, we create our triangle by simply using the polygon element and define three points relatively in pixels. The polygon element has its own canvas and therefore the points have to be defined on this canvas, but afterwards the canvas of the polygon can be positioned everywhere on the canvas of the whole visualisation. Because the polygon looks different depending in what order the points of it are connected with each other, we need also an index for every point of the polygon which specifies the order in which the points are connected. Thus, Fact (5.5) defines the first point of the polygon, the left lower point of the triangle, at position $(0, 20)$, Fact (5.6) defines the middle upper point, and Fact (5.7) defines the right lower point of the triangle. If these points are connected in the given order, one obtains a triangle which can then be relative positioned on the editor pane. The last part of the visualisation program uses relative positioning literals to draw the house with the sun. Of course, the sun must be the topmost element, which is defined by Facts (5.9) and (5.10), whereas the house must be under the roof which is taken care of by Fact (5.8). Finally, we want to draw the sun as the leftmost element and thus are using Facts (5.11) and (5.12).

The visualisation of this program can be found in Figure 5.1.

5.2 Maze generation

The problem description of the maze generation problem, along with its specification as an ASP program and a simple visualisation, was already given in Section 3.2. Here, we want to give a more advanced visualisation of this problem as well as to discuss more features of `Kara` by explaining the visualisation. We want to use a grid to visualise the maze and afterwards fill the grid with either an empty element, a wall, an entrance, or an exit. First, let us have a look at the important predicates for the visualisation: The output predicates are $wall/2$ as well as $empty/2$, but for our visualisation we are also interested in $entrance/2$ and $exit/2$. Furthermore, the encoding also includes two more predicates which are of significance for us, namely $maxcol/1$ and $maxrow/1$, which tell us the size of our grid in the visualisation. With this information about the output predicates, we can already start writing our visualisation program.

To begin with, we define our grid:

$visgrid(maze, ROW, COL, ROW*20+5, COL*20+5) :- maxcol(COL), maxrow(ROW), visposition(maze, 0, 0, 0).$

The first rule defines a grid with identifier $maze$ and row count ROW as well as column count COL, and a height of $ROW * 20 + 5$ and width of $COL * 20 + 5$. We multiply the row count by 20 because that will be the height of one rectangle respectively one image which we want to use to fill the grid. At the end, we add 5 pixels to the whole width because we also have to consider the border of the elements in the grid. We make the same for the width of the grid as for the height.

After defining the grid, the next task is to define the elements which we want to place inside the grid. Thus, we first define the empty cells as well as the cells representing the walls:

$$visrect(wall, 20, 20). \tag{5.13}$$
$$visbackgroundcolor(wall, black). \tag{5.14}$$
$$visrect(empty, 20, 20). \tag{5.15}$$

Fact (5.13) has identifier $wall$ and defines the graphical representation for all walls with a width of 20 and a height of 20 pixels. Fact (5.14) is then used to set for all rectangles representing a wall the background colour to black. The same as for walls is done for empty cells: they are represented by a rectangle with a white background colour and also have width and height of 20 pixels. We do not need to set the background colour to white in this case as this is the default background colour for all graphical elements which can be created.

After having defined how a wall and empty cells in the grid are represented by our visualisation, the next step is to define how the entrance and the exit should look like. This is done as follows:

$visimage(entrance, \text{“}wlp11/vis/img/entrance.jpg\text{”}),$
$visscale(entrance, 17, 17),$
$visimage(exit, \text{“}wlp11/vis/img/exit.png\text{”}),$
$visscale(exit, 17, 17).$

For representing the entrance and the exit, we want to use images which we saved in our project folder. The user can take either an absolute or a relative image path, where the root folder for the relative image path is the project root, whether the visualisation file is in a subfolder or not. Note that the second argument of the $visimage/2$ predicate is a string because it can contain special characters that should not be interpreted by the solver. The $visscale/2$ predicate is used to scale images, where we scale both images in our case to a width and height of 18 pixels. We do not take the full 20 pixels as in the case of the rectangles for walls and empty fields, because our images have no border.

Next, we define which element to place in which cell of the grid. To this end, we use the following rules:

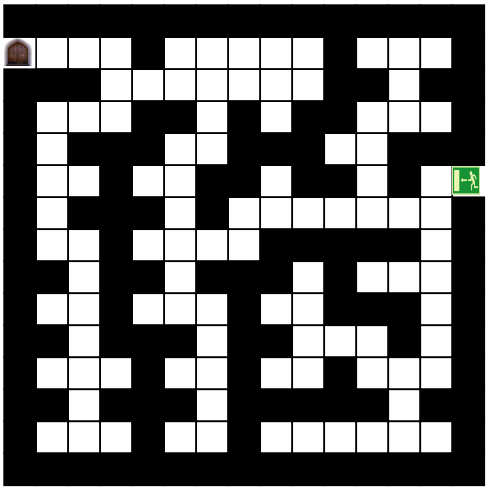

Figure 5.2: An advanced visualisation of the maze generation example with walls, empty fields, an entrance, and an exit.

$$
\begin{aligned}
visfillgrid(maze, empty, Y, X) :- \ & empty(X, Y), \\
visfillgrid(maze, wall, Y, X) :- \ & wall(X, Y), \\
visfillgrid(maze, entrance, Y, X) :- \ & entrance(X, Y), \\
visfillgrid(maze, exit, Y, X) :- \ & exit(X, Y).
\end{aligned}
$$

The predicate $visfillgrid/4$ is used to fill a grid, where

- the first parameter is the identifier of the grid which should be filled,
- the second parameter is the identifier of an element which should be placed on the grid,
- the third parameter defines the column of the grid, and
- the fourth parameter defines the row of the grid.

We need four rules in order to fill the maze with all elements; one rule for each element.

Finally, we add four more facts in order to enable the modification of the visualisation; in particular, we want to change elements in our maze (e.g., change an empty field into a wall):

$$
\begin{aligned}
&vispossiblegridvalues(maze, wall), \\
&vispossiblegridvalues(maze, empty), \\
&vispossiblegridvalues(maze, entrance), \\
&vispossiblegridvalues(maze, exit).
\end{aligned}
$$

The facts allow that all cells in our maze can be changed with either a wall, an empty field, an entrance, or an exit.

Putting all parts together and visualising a maze interpretation with this visualisation program, we get the visualisation shown in Figure 5.2.

Assume now that we want to change the visual representation of the maze and afterwards infer the corresponding interpretation I' of the modified visualisation interpretation I'_v. To this end, we perform the steps shown in Figure 4.7. When the user right clicks on some cell of the maze, the context menu for operations are shown. It contains the following entries:

- The delete entry is only shown if the element on which the user clicked is annotated in the visualisation answer set with the predicate $visdeletable/1$.
- The "edit properties" action is only activated if the selected element has some properties marked as changeable by the visualisation answer set.
- The next entry is the one which will be used to change the element in the maze from a wall to an empty field.
- "Show interpretation" puts the visualisation answer set to the `SeaLion` visualisation view, and the "infer interpretation" action will be discussed later on.
- The last two entries are for exporting the current visualisation into SVG format. Either the whole visualisation can be exported or only the currently selected parts of it.

If we click on the "change element" button, a dialog opens and shows which elements can be inserted into a grid on the selected position and the current element be removed. The element names shown in this dialog correspond to the elements which we specified via the $vispossiblegridvalues/2$ predicate. Let us, for example, change the element on an empty field and move forward to step three where we can see that the wall on the second row and the fifth column is now empty. Then we want to infer the corresponding interpretation to the modified visualisation interpretation and thus we press "infer interpretation" on the context menu which automatically infers the interpretation without prompting for any user input. If the abduction was successful, the user finds the inferred interpretation in the interpretation view of `SeaLion` under the name "ABDUCTION". The other entry, "infer interpretation (specify domain)", automatically computes the domain for the inferring process and allows the user to edit the computed domain before constructing the whole abduction program; see Section 3.2 for details when this is necessary.

5.3 Graph colouring

The following example, the *graph colouring problem*, illustrates how `Kara` automatically calculates the layout of a graph such that the user may not worry about positioning every node and edge of the graph.

The graph colouring problem is the following task: Given a graph, i.e., a collection of nodes and a symmetric, binary relation on nodes, and a set of n colours, assign each node a colour such that any two nodes that are linked together have not the same colour.

First, let us have a look at a domain program solving this problem (in `Gringo` syntax) for $n = 3$:

$$\#const \quad n = 3. \tag{5.16}$$
$$1\ \{color(X, 1..n)\}\ 1 :- node(X). \tag{5.17}$$
$$:- edge(X, Y), color(X, C), color(Y, C). \tag{5.18}$$

Rule (5.17) assigns every node exactly one colour and Rule (5.18) checks that all adjacent nodes have different colours. The input instance for our example is the following:

$$node(1..6). \tag{5.19}$$
$$edge(3, 5).\ edge(3, 6).\ edge(3, 1).\ edge(3, 4).\ edge(5, 6).\ edge(5, 2). \tag{5.20}$$
$$edge(5, 4).\ edge(2, 4).\ edge(2, 1).\ edge(2, 6).\ edge(4, 1). \tag{5.21}$$

The output of the domain program is the predicate $color/2$, where the first argument is the identifier of the node and second is the colour of the node given as an integer. Furthermore, we retrieve information about the graph via the predicates $node/1$ and $edge/2$, for which we need to define graphical elements for drawing the graph. With this information, we can define the following visualisation program:

$$visgraph(g), \tag{5.22}$$
$$visisnode(X, g) :- node(X), \tag{5.23}$$
$$visellipse(X, 20, 20) :- node(X), \tag{5.24}$$
$$vislabel(X, l(X)) :- node(X), \tag{5.25}$$
$$vistext(l(X), X) :- node(X), \tag{5.26}$$
$$visfontstyle(l(X), bold) :- node(X), \tag{5.27}$$
$$visbackgroundcolor(X, COLOR) :- node(X), color(X, COLOR), \tag{5.28}$$
$$visconnect(f(X, Y), X, Y) :- edge(X, Y). \tag{5.29}$$

Fact (5.22) defines a graph with identifier g which is later used to assign nodes to a specific graph. Rule (5.23) defines every input node as a node in the visualisation using predicate $visisnode/2$, where the first parameter is the identifier of the node and second one references the identifier of a graph. Note that predicate $visisnode/2$ defines nothing for rendering the element but that this element with its identifier belongs to a specific graph. The form with which the element is rendered is defined as usual by predicates like $visrect/2$ and $visellipse/2$ such that it is necessary for every node defined with the predicate $visisnode/2$ to define also the form with which it is rendered. Rule (5.24) defines that every node is rendered as an ellipse (in our specific case, it is a circle), and Rules (5.25), (5.26), as well as (5.27) are used to assign a label to each node. The predicate $vislabel/2$ references with its first argument the element to which the label is assigned and the second argument references a text node which defines the content of the label. The predicate $vistext/2$ defines with its second argument the content of label, which can be a string, and therefore it can have any possible text value and is not interpreted by the solver. To make the label for the nodes better visible, we want to draw it with bold font style which is

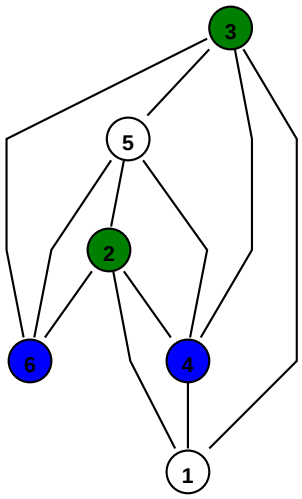

Figure 5.3: Custom visualisation of the graph colouring example.

achieved by using predicate $visfontstyle/2$, which is similar to $viscolor/2$ because the second argument is also a predefined constant by `Kara`, meaning that values other than the ones defined by `Kara` are not accepted. Rule (5.28) sets the background colour of each node in accord with the colour assignment we get from the answer set of the domain program. Here, we have the nice feature that predicate $visbackgroundcolor/2$ does not only take names of colours as arguments but also natural numbers. This is because `Kara` internally maintains a colour table and the colour corresponding to a natural number is then chosen for visualisation. Thus, the numbers as outputted by the domain program may not be mapped to colour names. Rule (5.29) creates the edges of the nodes with predicate $visconnect/3$, where the first argument is a newly created identifier using a function symbol having as argument the two nodes which are connected with each other. The second argument is the source of the edge and the last one is the target of the edge. Note that there are also further predicates which can set the source and the target decoration of an edge (see Table 3.1).

As stated above, we do not make use of any absolute positioning; the whole graph is automatically positioned by `Kara`. The final visualisation can be found in Figure 5.3.

5.4 15-puzzle

The problem of solving the *15-puzzle*, taken from the Second ASP competition [23], is defined as follows: Given a 4×4 grid containing numbers 1 to 15 and one blank, the goal is to arrange the numbers from their initial configuration to a goal configuration by swapping one number at a time with its adjacent blank position. Let (x, y) be the coordinates of a number on the grid and (i, j) those of the blank. Then, (x, y) and (i, j) are adjacent if $|x - i| + |y - j| = 1$.

Following the guidelines of the Second ASP competition, encodings of the problem use the following input and output predicates (different encodings for the 15-puzzle can be found on the web page of the competition): Predicate $entry/1$ defines a single field on the 15-puzzle (the

$$visrect(b(ENTRY),20,20) :- \ entry(ENTRY). \tag{5.30}$$
$$vislabel(b(ENTRY),bt(ENTRY)) :- \ entry(ENTRY). \tag{5.31}$$
$$vistext(bt(ENTRY),ENTRY) :- \ entry(ENTRY). \tag{5.32}$$
$$visfontsize(ID,10) :- \ vistext(ID,\_). \tag{5.33}$$
$$visgrid(g(T),4,4,85,85) :- \ time(T). \tag{5.34}$$
$$visfillgrid(g(0),b(ENTRY),X,Y) :- \ in0(X,Y,ENTRY). \tag{5.35}$$
$$visleft(g(X),g(Y)) :- \ visgrid(g(X),\_,\_,\_,\_), \tag{5.36}$$
$$visgrid(g(Y),\_,\_,\_,\_), Y = X+1. \tag{5.37}$$
$$visfillgrid(g(T),b(ENTRY),X,Y) :- \ \text{not } move(TB,X,Y), \tag{5.38}$$
$$T = TB+1, \tag{5.39}$$
$$visfillgrid(g(TB),b(ENTRY),X,Y), \tag{5.40}$$
$$ENTRY! = 0, time(T). \tag{5.41}$$
$$visfillgrid(g(T),b(ENTRY),X,Y) :- \ \text{not } move(TB,X,Y), \tag{5.42}$$
$$T = TB+1, \tag{5.43}$$
$$visfillgrid(g(TB),r(TB,ENTRY),X,Y), \tag{5.44}$$
$$ENTRY! = 0, time(T). \tag{5.45}$$
$$visfillgrid(g(T),b(0),X,Y) :- \ move(TB,X,Y), T = TB+1. \tag{5.46}$$
$$visfillgrid(g(T),r(T,ENTRY),X,Y) :- \ T = TB+1, visfillgrid(g(TB),b(0),X,Y), \tag{5.47}$$
$$move(TB,X1,Y1), \tag{5.48}$$
$$visfillgrid(g(TB),b(ENTRY),X1,Y1). \tag{5.49}$$
$$visrect(r(T,ENTRY),20,20) :- \ visfillgrid(\_,r(T,ENTRY),\_,\_). \tag{5.50}$$
$$vistext(rt(T,ENTRY),ENTRY) :- \ visrect(r(T,ENTRY),\_,\_). \tag{5.51}$$
$$vislabel(r(T,ENTRY),rt(T,ENTRY)) :- \ visrect(r(T,ENTRY),\_,\_). \tag{5.52}$$
$$viscolor(rt(T,ENTRY),red) :- \ vistext(rt(T,ENTRY),\_). \tag{5.53}$$

Figure 5.4: Visualisation program of the 15-puzzle.

numbers 1 to 15 are defined with this predicate) where 0 denotes the blank. The predicate $pos/1$ defines the possible values for the x and y position of the fields. The maximum number of steps is given by predicate $maxtime/1$ and for every time point one atom exists with predicate $time/1$. Finally, predicate symbol $in0/3$ has as its arguments the (x, y) position and the corresponding field as its last argument.

The output of the solution is given by atoms with predicate symbol $move/3$ having the time stamp as its first argument and the row and column position of the element which should be moved with the blank as the second and third argument, respectively. At the end, either the maximum time is reached or the problem instance is solved.

The intent of the visualisation is to show every step for obtaining the solution, i.e., every move which has to be taken in order to reach the problem solution that all numbers are in ascending order on the puzzle. Thus, we use multiple grids in a row on the visualisation, employing

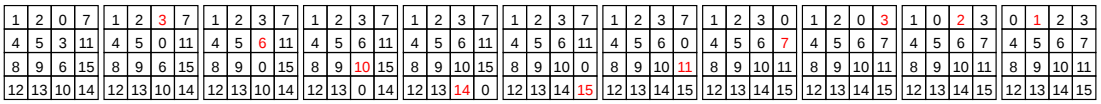

Figure 5.5: Visualisation of the 15-puzzle.

relative positioning to keep the movements in correct order left to each other. The visualisation program for the 15-puzzle is depicted in Figure 5.4.

For every field of the 15-puzzle, we create one rectangle which is done by Rule (5.30). These rectangles are a label in Rule (5.31) with their number as text representation in Rule (5.32). For every text on our visualisation, we set the font size to 10 pixels which is done by Rule (5.33).

After defining the representation of every single field in our graphical representation, we define the grids such that for every time point we use a single grid representing the current state of the puzzle at this time point. This is done by Rule (5.34), where we define for each time point a grid with four columns and four rows, and with a width and height of 85 pixels. We use the function symbol $g/1$ in conjunction with the number of the time point to create the identifier of the grid. Rule (5.35) constructs the initial grid $g(0)$ from the input predicate $in0/3$ and refers to the rectangles visualising the fields of the puzzle with function symbol $b/1$. With Rule (5.36) we define the relative positioning of our grids using predicate symbol $visleft/2$. Rule (5.38) copies all fields which were not moved to the grid of the next time point using predicate symbol $visfillgrid/4$. Of course, the blank is not copied because it is moved on every time point. Rule (5.42) performs exactly the same copy task as the previous rule but only for red-coloured labels because they have another identifier and should now be converted to a black label as they have not changed in this time step. Rule (5.46) is used to move the blank field in the new grid to the position where the moved element was. This can be seen because of the function $b(0)$, where 0 is referred to the blank as stated above. Now, the last step is to move the entry, which is defined by predicate $move/3$, to the position where the blank was. This task is done by Rule (5.47), where we use as in the previous rule also the function $b(0)$ to refer to the blank. The last rules are only used to define the red labels for those entries which have changed at some time point. We introduce two function symbols to this end, namely $r/1$ to refer to the rectangle and $rt/1$ to refer to the text associated to the rectangle. Rule (5.50) is used to create the rectangle with the new function symbol whereas Rule (5.51) is used to create the text for this rectangle. Moreover, Rule (5.52) defines the association of the text to the rectangle using predicate symbol $vislabel/2$. Finally, Rule (5.53) is used to set the colour of the new created text to red to indicate that this entry has moved in the current time point.

The visualisation of the 15-puzzle using this visualisation program is depicted in Figure 5.5.

CHAPTER 6

Related work

Kara follows the approach of the two tools ASPVIZ [22] and IDPDraw [21], which we examine in what follows. To this end, we give a simple example visualisation of a maze which we use to compare with equivalent visualisations written with ASPVIZ and IDPDraw. Recall that the problem description for the maze generation problem is given in Section 3.2. The visualisation program written in Kara looks as follows:

$$visrect(f(X,Y),20,20) :- \ wall(X,Y), \tag{6.1}$$
$$visbackgroundcolor(f(X,Y),black) :- \ wall(X,Y), \tag{6.2}$$
$$visposition(f(X,Y),X*20,Y*20,0) :- \ wall(X,Y). \tag{6.3}$$

Rule (6.1) creates a rectangle for every wall in the maze encoding and Rule (6.2) sets the background colour of the walls to black. Finally, Rule (6.3) sets the position of the rectangles according to the logical position encoded in the answer set. The visualisation output of this short visualisation program is given in Figure 3.1.

6.1 ASPVIZ

ASPVIZ was developed at the University of Bath and can produce graphical representations from answer sets. In ASPVIZ, a special visualisation program V is used to map the atoms of the answer set to a graphical representation. V is an answer-set program and contains dedicated visualisation predicates, which is also implemented in Kara this way. However, the visualisation predicates of the two tools are different. ASPVIZ uses *brushes* to define properties on elements (e.g., background colour, font weight, etc.), whereas Kara assigns properties directly to the elements themselves. Both ASPVIZ and Kara use identifiers to refer to the brush (in ASPVIZ) respectively the element (in Kara). In ASPVIZ, the created brushes may be assigned to some elements, whereas in Kara, the identifier of the elements are used to assign special visualisation properties of the elements. Furthermore, Kara supports more visualisation predicates

$$scale\_canvas(7,7). \tag{6.4}$$
$$color(white, rgb(255,255,255)). \tag{6.5}$$
$$color(black, rgb(0,0,0)). \tag{6.6}$$
$$brush(bwhite). \tag{6.7}$$
$$brush\_color(bwhite, white). \tag{6.8}$$
$$brush\_width(bwhite, 1). \tag{6.9}$$
$$brush(bblack). \tag{6.10}$$
$$brush\_color(bblack, black). \tag{6.11}$$
$$brush\_width(bblack, 1). \tag{6.12}$$
$$draw\_rect(bwhite, p((X-1)*20,(Y-1)*20),20,20) :- \ empty(X,Y). \tag{6.13}$$
$$fill\_rect(bblack, black, p((X-1)*20,(Y-1)*20),20,20) :- \ wall(X,Y). \tag{6.14}$$

Figure 6.1: A visualisation program for maze generation in ASPVIZ.

than ASPVIZ to make it easier for the user to define the graphical visualisation of the problem. Examples are the relative positioning of elements, definition of graphs, where the layout is calculated automatically, as well as grids, where the content of each cell can be defined separately. ASPVIZ supports also the creation of animations by using a special predicate $frame/1$, which takes a natural number as argument. The visualisation program may create many answer sets which are ordered by ASPVIZ according to their frame number and are then displayed in this order by the animation. ASPVIZ exports every graphical visualisation into SVG files and therefore does not support the editing of its visualisations, whereas Kara displays the graphical visualisation in a graphical editor where it can be manipulated. Kara also supports exporting the created graphical representation into SVG, even when only parts of the graphical representation are selected. Kara is integrated in Eclipse and therefore can be executed graphically, whereas ASPVIZ is a command-line tool. Also, the generic visualisation of Kara is not available in ASPVIZ. This kind of visualisation renders a hypergraph of the answer-set program written by the user according to its syntactic structure, which cannot be edited.

Both tools are written in Java, where Kara is written within the Eclipse plugin framework (PDE), and thus are platform independent. Kara supports also a z-index defining which element is shown in case two elements are overlapping. This is not implemented in ASPVIZ, but there is a new version, ASPVIZ-3D, which supports 3-dimensional objects and animations of them. ASPVIZ supports colours with their RGB value, where a function symbol $rgb/3$ is used for representing colours, whereas in Kara full names (e.g., red, blue, green etc.) or integers (internally mapped to the available colours in Kara) can be used to this end. ASPVIZ uses no prefix to their visualisation predicates, which can be problematic due to possible name clashes, while Kara uses the prefix *vis* to all its available visualisation predicates.

An example visualisation program for the maze generation problem using ASPVIZ syntax is depicted in Figure 6.1. Here, the visualisation program looks slightly different. First, we have to scale the canvas[1] in order to correctly fit the maze into the graphical output (cf. Fact (6.4)).

[1]The area where the output fits in is called *canvas*.

Next, the RGB values for the two colours black and white are defined by Facts (6.5) and (6.6). In contrast to Kara, in ASPVIZ we do not set properties on objects but define brushes, which set the line style, colour, etc., of the objects. Here, we define two brushes, a black one and a white one, where we need the white brush for drawing the empty fields and the black brush for drawing the walls. Fact (6.7) defines first that the constant $bwhite$ is referred to as a brush, and Fact (6.8) defines that the colour of the brush is white. With Fact (6.9) we state that the line width associated to the brush has a width of 1, the thinnest value. The same is done for the black brush which is defined by Facts (6.10) to (6.12). Rule (6.13) renders an empty rectangle with white border colour for every empty cell in the input interpretation, whereas Rule (6.14) states that for every wall in the input interpretation a black rectangle is drawn.

6.2 IDPDraw

IDPDraw also uses a visualisation program with special dedicated predicates to create the visualisation of an answer set of a given domain program. However, IDPDraw does not support parsing the output of various solvers directly, thus their output must be post-processed (e.g., the tool sed[2] may be used in order for IDPDraw to correctly parse the output). IDPDraw has another kind of identifiers: As in Kara, IDPDraw uses identifiers for elements, but they are constructed with multiple arguments at the beginning of the predicate instead of using only the first argument as is realised in Kara[3]. IDPDraw neither supports elements like graphs and grids automatically nor relative positioning. However, graphs and grids can be constructed by using other (simpler) elements, but it is much more difficult than in Kara. As ASPVIZ, also IDPDraw supports animations of visualisations, which are played in the graphical user interface of the tool instead of constructing an animated SVG. The animations are produced by using the predicates with a postfix "$\_t$" and an additional time argument in the respective atom.

IDPDraw is written in C++ based on the QT framework [36] and is thus platform independent, because QT offers a cross-platform application development. In contrast to Kara, IDPDraw only supports the use of RGB colours. Both tools offer support for a z-index, defining which elements are shown if two or more of them are overlapping. The editing of visualisations and inferring the corresponding interpretation of the graphical representation is not supported by the user interface of IDPDraw as it is supported by Kara. Also, the generic visualisation feature of Kara is not supported by IDPDraw.

The following program is a visualisation for the maze generation problem in IDPDraw:

$$idpd\_polygon(4, R, C, 0, 0, 1, 0, 1, 1, 0, 1) :- \ wall(R, C), \tag{6.15}$$
$$idpd\_xpos(R, C, R-1) :- \ row(R), col(C), \tag{6.16}$$
$$idpd\_ypos(R, C, C-1) :- \ row(R), col(C), \tag{6.17}$$
$$idpd\_color(R, C, 0, 0, 0) :- \ wall(R, C). \tag{6.18}$$

In IDPDraw, the background colour is automatically set to white, so it is the best strategy to create polygons with black background colour to render the walls. In Rule (6.15), the polygons

[2] http://unixhelp.ed.ac.uk/CGI/man-cgi?sed.

[3] For generating elements, function symbols may be used.

are created to represent walls. The first parameter of $idp\_polygon$ is the number of vertices of the polygon, the second and third build the identifier of the object together. Note that the identifier can be as long as the user wants and thus the predicates can have variable length. The next arguments of $idp\_polyogon$ specify the relative position of each corner of the polygon. As can be seen, a rectangle is created. In contrast to `Kara`, `IDPDraw` uses a separate predicate for absolute positioning of the elements for every axis. The x-position is set by Rule (6.16) and the y-position is set by Rule (6.17). The visibility of overlapping elements is controlled in `IDPDraw` via predicate $idpd\_depth$ which is equivalent to the z-axis in `Kara`. Rule (6.18) is used to set a black background colour on all wall elements in the visualisation.

6.3 Lonsdaleite

Another visualisation tool is `Lonsdaleite` [37], which is specialised to visualise graphs respectively graph structures. Thus, it only supports rendering a problem in a graph structure, but not other elements as it is possible in `ASPVIZ`, `IDPDraw`, and `Kara`. `Lonsdaleite` is a very lightweight tool, consisting only of one Python source file, as it only depends on Python installed on the target system. The tool renders the graph structure with the support of `graphviz`[4], where each atom interpreted by `Lonsdaleite` is mapped to some `graphviz` setting or property and nearly all `graphviz` settings are supported. Atoms belonging to the dedicated visualisation predicates of `Lonsdaleite` are prefixed with "*graphviz_*", which helps avoiding name clashes with the domain program. Actually, even the `graphviz` library need not be available on the user's system, because `Lonsdaleite` offers the possibility to render the visualisation in a web-browser using the `graphviz` chart API of Google.

6.4 APE

There exists also an earlier visualisation approach of answer-set programs as a *dependency graph*. It is part of an answer-set programming IDE toolset implemented as an Eclipse plugin, called `APE` [38]. `APE` uses also the graphical editor framework of Eclipse and GEFs default user interface framework `Draw2d` for the visualisation itself. To automatically generate the graph and its layout, `APE` relies on the graph drawing component of the `Draw2d` library, as is also implemented in `Kara`.

6.5 DPVis

`DPVis` [26] is a tool for visualising SAT instances, which displays the SAT problems as variable interaction graphs as well as resolution graphs. `DPVis` does not only show static visualisations of SAT instances, but also shows animations of the change of the problem structure during the run of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [39]. `DPVis` is written in `Java` and relies on the `yWorks`[5] graph-layouting framework (recall that `Kara` uses the

[4] `http://www.graphviz.org/`.
[5] `http://www.yworks.com`.

Draw2d graph-layouting framework). Furthermore, DPVis uses the MiniSat solver[6] to calculate the solutions to the SAT instances. The user is able to display a partial search tree, which can be navigated freely, and may zoom as well as set the states of various variables manually.

There is also a variant of DPVis, called 3DVis [26], which supports three-dimensional visualisations of SAT instances. 3DVis is implemented in C++ using OpenGL and uses the force-directed graph-drawing algorithm [40] for the three-dimensional visualisation of the SAT instances. DPVis is platform independent whereas 3DVis only supports Windows and MacOS.

6.6 Alloy

Another tool for visualisation of finite structures is Alloy [25], which features a kind of first-order language. The user is able to define objects represented as signatures, which can have properties (e.g., relationships to other objects). By using facts and assertions, the user is able to define constraints on the structure. Moreover, predicates allow to define instances of the objects which can be executed afterwards. Signatures are represented as nodes and their relationships are represented as edges. Thus, the visualisation itself is encoded in the user-defined structure itself and there is no need for writing any visualisation program. However, this relies on the fact that the visualisation of those structures is limited to very basic graph-layouts.

If an instance is executed by Alloy, it is checked for consistency. If it is consistent, a model will be given, otherwise a counterexample is provided. Alloy does not only support graphical visualisations in its own editor but also allows the user to export the visualisation in DOT or XML language as well as PNG and PDF.

Further features include syntax highlighting, modularisation of source files, as well as the ability to configure the SAT solver in the background. The graphical visualisation can be customised (but not edited) by the user (e.g., changing the layout or projection of objects and properties). An example of an Alloy module is the following [25]:

```
module tour/addressBook2a

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }

sig Alias, Group extends Name { }

sig Book {
addr: Name->Target
}

pred show [b:Book]   { some b.addr }

run show for 3 but 1 Book
```

[6] http://minisat.se/.

In `Alloy`, one can define modules and use them in terms of the keyword `import`. In this example, a new module is created with the name `tour/addressBook2a`. Furthermore, types are defined using the keyword `sig`, and subtypes can be created by extending an existing signature. Attributes of the signatures can be defined within the brackets of the type definition. Predicates defined with keyword `pred` are instances of the signatures. In the example above, the keyword `some` is used, which refers to a quantifier and means one or more. The `run` command is used to execute predicate or functions.

CHAPTER 7

Conclusion

In this work, we presented the tool `Kara` for visualising and visual editing of answer sets. The visualisation of answer sets is an important task because the text representation of answer sets is very cumbersome for humans to be interpreted and checked for correctness. We believe that visualisations as presented in this work are helpful for a better understanding of the solver output. Indeed, in most cases, the programmer is able to immediately see whether a solution is correct or not.

It is a very important property for visualisation tools to have a very easy and fast way to write visualisation programs such that visualising answer sets is not a time-consuming task for the user of these visualisation tools. The programmer should focus on the implementation of the domain program but not on writing visualisations. `Kara` aims to fulfill this by offering a wide range of visualisation predicates, the use of identifiers and relative positioning, and the use of automatically positioned elements. As presented in this thesis, if the user only wants to write a short visualisation to the problem at hand, he or she can do so, as seen, e.g., by the visualisation of the maze generation problem which only needs three rules to obtain a visualisation.

The use of identifiers allows to directly refer to each element in the program and set properties on it such that the user may not give details about all parameters on element creation. Relative positioning and the predicates for graphs and grids help the user in positioning the elements in the visualisation without the need to worry about their absolute positions.

A new feature currently not implemented in related tools is the possibility to *edit a visualisation*, which paves the way for new methods to debug and test answer-set programs. In case of visualisations of incorrect, i.e., unintended, answer sets, the visualisation can be edited graphically and then a corresponding, corrected interpretation computed. On the other hand, visualisations can also be used for generating interpretations which can be of interest if one needs answer sets for testing post-processing tools or in modular programming to test a module before other required modules are available.

`Kara` is implemented within the Eclipse framework, one of the currently most used IDEs for `Java` programming, and depends on the `SeaLion` system. Eclipse offers many plugins, and thus many programmers are familiar with this IDE, which is in turn beneficial for `Kara`. Also,

looking at portability of visualisations, `Kara` offers an SVG export function for all graphical problem representations.

Although `Kara` offers a convenient way to realise visualisations, it has to be evaluated further as far as usability is concerned. Moreover, concerning future work, a possibility would be the creation of a command-line version of `Kara` without the editing feature in order to get fast visualisations without the use of the Eclipse IDE and which would make it possible to have batch visualisations to visualise multiple answer sets of a program. Also, for some domain programs, it might be beneficial to visualise both the input as well as the output, and perhaps bind both visualisations to each other such that if one is modified the other one changes accordingly. Another consideration is the implementation of predicate signatures which would ease the domain computation in the abduction program. However, this should be implemented in the `SeaLion` core as this feature can also be used by the core itself as well as by other plugins integrated in the `SeaLion` system.

Bibliography

[1] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and the 25th Workshop on Logic Programming* (*WLP 2011*), pages 152–164, 1843-11-06, 2011. INFSYS Research Report, Technische Universität Wien.

[2] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[3] Riccardo Rosati. $\mathcal{DL}$+*log*: Tight integration of description logics and disjunctive datalog. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 68–78. AAAI Press, 2006.

[4] Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic composition of melodic and harmonic music by answer set programming. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), volume 5366 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2008.

[5] Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. An ASP-based system for e-tourism. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), volume 5753 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 2009.

[6] Esra Erdem and Elisabeth R. M. Tillier. Genome rearrangement and planning. In *Proceedings of the 20th National Conference on Artificial Intelligence* (*AAAI 2005*), pages 1139–1144, 2005.

[7] Esra Erdem, Vladimir Lifschitz, and Donald Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558, 2006.

[8] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), pages 260–265. Springer, 2007.

[9] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[10] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10(4-6):513–529, 2010.

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

[12] Martin Brain and Marina De Vos. Debugging logic programs under the answer set semantics. In *Proceedings of the 3rd International Workshop on Answer Set Programming* (*ASP 2005*), volume 142 of *CEUR Workshop Proceedings*, 2005.

[13] Tommi Syrjänen. Debugging inconsistent answer set programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning* (*NMR 2006*), pages 77–84, Lake District, UK, May 2006.

[14] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence* (*ECAI 2010*), pages 951–956, 2010.

[15] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Random vs. structure-based testing of answer-set programs: An experimental comparison. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer, 2011.

[16] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research*, 35:813–857, 2009.

[17] Tomi Janhunen. Modular equivalence in general. In *Proceedings of the 18th European Conference on Artificial Intelligence* (*ECAI 2008*), pages 75–79. IOS Press, 2008.

[18] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence ECAI* (*ECAI 2006*), pages 412–416. IOS Press, 2006.

[19] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. "That is Illogical Captain!" – The debugging support tool spock for answer-set programs: System description. In *Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming* (*SEA 2007*), pages 71–85, 2007.

[20] Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. spock: A debugging support tool for logic programs under the answer-set semantics. In *Proceedings of the 21st Workshop on* (*Constraint*) *Logic Programming* (*WLP 2007*), pages 258–261. Technical Report 434, Julius-Maximilians-Universität Würzburg, Institut für Informatik, 2007.

[21] Johan Wittocx. IDPDraw: A tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation`, 2009.

[22] Owen Cliffe, Marina De Vos, Martin Brain, and Julian A. Padget. ASPViz: Declarative visualisation and animation using answer set programming. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), volume 5366 of *Lecture Notes in Computer Science*, pages 724–728. Springer, 2008.

[23] Second Answer-Set Programming Competition. `http://dtai.cs.kuleuven.be/events/ASP-competition/`, 2009.

[24] Ehud Y. Shapiro. *Algorithmic Program Debugging*. PhD thesis, Yale University, New Haven, CT, USA, May 1982.

[25] Daniel Jackson. *Software Abstractions – Logic, Language, and Analysis*. MIT Press, 2006.

[26] Carsten Sinz. Visualizing SAT Instances and Runs of the DPLL Algorithm. *Journal of Automated Reasoning*, 39:219–243, August 2007.

[27] Johannes Oetsch, Jörg Pührer, and Hans Tompits. 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*), pages 141–151, 1843-11-06, 2011. INFSYS Research Report, Technische Universität Wien.

[28] The Eclipse Foundation. Eclipse Graphical Editing Framework. `http://www.eclipse.org/gef/`.

[29] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web*, pages 40–110, 2009.

[30] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming* (*ICLP 1988*), pages 1070–1080, 1988.

[31] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[32] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 402–436. Springer, 2002.

[33] Mario Alviano. The maze generation problem is NP-complete. Proceedings of the 11th Italian Conference on Theoretical Computer Science (ICTCS 2009), 2009.

[34] Third Answer-Set Programming Competition. `https://www.mat.unical.it/aspcomp2011/`, 2011.

[35] W3C. SVG working group. `http://www.w3.org/Graphics/SVG/`.

[36] Nokia. QT Framework. `http://qt.nokia.com/`.

[37] Lonsdaleite. `https://github.com/rndmcnlly/Lonsdaleite`, 2011.

[38] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* environment. In *Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming* (*SEA 2007*), pages 101–115, Tempe, AZ, USA, 2007.

[39] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[40] Chris Walshaw. A Multilevel Algorithm for Force-Directed Graph-Drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.