FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ARVis - Programmer's Guide

Visualization of relationships between answer sets

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Thomas Ambroz
Matrikelnummer 0925772

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Univ.Ass. Dipl.-Ing. Günther Charwat

Wien, 29.06.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

ARVis - Programmer's Guide

Visualization of relationships between answer sets

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Thomas Ambroz
Registration Number 0925772

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Univ.Ass. Dipl.-Ing. Günther Charwat

Vienna, 29.06.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Ambroz
1120 Wien, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig 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)

Abstract

Answer Set Programming (ASP) becomes more and more popular. The complex and high-performance features of ASP languages make it easy to solve problems in non-deterministic polynomial time. The result of such problems often consists of many different solutions. Some solutions (answer sets) could be similar, they could have some common properties or interrelate. Such relationships could be very interesting. Therefore it is helpful to use this aspect and create a graph visualization, containing the relationships between these answer sets. This Bachelor's thesis comprises such a visualization process by introducing the software application called *ARVis (Answer Set Relationship Visualizer)* and its *Programmer's Guide*. Therefore, the structure, components, libraries, classes and interfaces of *ARVis* are described including source code examples and explanations for the better understanding of the complete process. The final software provides a simple user interface and supports ASP solvers like DLV, clingo and a customized combination of grounder and solver (like gringo and clasp). The conditions whether two answer sets correlate or not are defined by a separate ASP program. The computed relationships between the answer sets are represented by a simple and interactive graph.

Kurzfassung

Answer Set Programming (ASP) wird heutzutage immer bedeutsamer. Die komplexen und leistungsfähigen Funktionen von ASP Programmiersprachen ermöglichen es, Probleme in nicht-deterministischer polynomialer Zeit zu lösen. Das End-Ergebnis dieser Probleme besteht häufig aus vielen unterschiedlichen Einzellösungen. Manche dieser Lösungen (Answer Sets) können sich ähneln, sie können gemeinsame Eigenschaften aufweisen oder miteinander in Beziehung stehen. Daher ist es hilfreich, diese gemeinsamen Eigenschaften und Beziehungen von Answer Sets mit Hilfe eines Graphen darzustellen. In dieser Bachelorarbeit wird solch ein Visualisierungsprozess vorgestellt, indem die Software *ARVis (Answer Set Relationship Visualizer)* beschrieben wird – in Form einer Anleitung aus Entwicklersicht (*Programmer's Guide*). Daher werden Aufbau, Komponenten, Java Bibliotheken, Klassen und Schnittstellen von *ARVis* beschrieben – zusätzlich mit vielen Quelltext Beispielen und Erklärungen für ein besseres Verständnis des gesamten Vorgangs. Die Anwendung enthält eine leicht bedienbare Benutzeroberfläche und unterstützt ASP Solver wie DLV, clingo und eine benutzerdefinierte Kombination von Grounder and Solver (wie beispielsweise gringo and clasp). Die Bedingungen, ob zwei Answer Sets in Beziehung stehen, werden durch ein eigenes ASP Programm definiert. Ein einfacher interaktiver Graph stellt letztendlich die berechneten Beziehungen der Answer Sets dar.

Contents

1 Introduction **1**

2 Answer Set Programming **3**
2.1 Background and application . . . 3
2.2 Syntax and semantics . . . 4

3 ASP-Solvers **7**
3.1 DLV . . . 7
3.2 Clingo (gringo + clasp) . . . 9
3.3 Specialization: User defined combination of grounder and solver . . . 11
3.4 Differences . . . 11
3.5 ASP Wrapper: Embedding ASP-Solver in independent external applications . . . 13

4 Software “ARVis” **15**
4.1 Background: The presentation of answer sets and their relationships . . . 15
4.2 Layout and structure of the software . . . 17
4.3 Interfaces & libraries . . . 19
4.4 Use of the wizard for program guidance . . . 20
4.5 Connection to ASP-Solver . . . 22
4.6 Parser: The processing of the result data . . . 29
4.7 Calculation and setting of relationships . . . 31
4.8 Presentation as graph . . . 33

5 Conclusion **37**

Bibliography **39**

CHAPTER 1

Introduction

Nowadays, Answer Set Programming (ASP) becomes more and more popular – for academic purposes as well as in industrial real-world applications. The complex and high-performance features of ASP languages make it easy to solve problems which are classified to be solvable in non-deterministic polynomial time (NP) or also in NP^{NP}. The result of such problems often consists of many different solutions, represented by simple plain text terms [5, 11, 15]. Therefore it could sometimes be difficult to interpret these results or to keep an overview. Additionally, complex problems can lead to complex solutions, consisting of quite a lot of terms, called facts. Due to this problem, some software solutions have been developed, helping to give a better overview by doing visualization processes based on the results of the ASP program. On the one hand, such a visualization could be done by representing each fact of an answer set as a visualization element. Further on, other graphical elements can be used to classify and group the facts. This way, each calculated solution would be organized to understand it in a better and simpler way. This visualization process could be realized with shapes, colors and text elements or with a simple graph by using nodes and edges. Such software applications are currently existing on the market and are still in development [13]. On the other hand, the visualization could have another main focus: Instead of considering every single solution (answer set) as closed entity, the collection of all calculated solutions can be considered as a whole. Some solutions (answer sets) could be similar, they could have some common properties or interrelate. Such relationships are very interesting in some application cases and therefore it is useful to gather this aspect of visualization too.

This bachelor's thesis introduces this new aspect of visualization in detail by providing a *Programmer's Guide* for a concrete software application called *ARVis (Answer Set Relationship Visualizer)*[1]. It implements such a visualization of answer sets. This application is a self-development of Thomas Ambroz and Andreas Jusits. It was developed for the bachelor's thesis

[1] http://www.dbai.tuwien.ac.at/proj/argumentation/vispartix/#ARVis

in cooperation with Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran and Univ.Ass. Dipl.-Ing. Günther Charwat.

The *Programmer's Guide* starts with an introduction to Answer Set Programming (ASP) in general. Therefore the background, history and application of ASP will be described and the some basics about syntax and semantics of Answer Set Programming will be mentioned.

The next chapter presents some specific ASP Solvers like *DLV* and *clingo* by explaining the functionality, fields of application and advantages [10, 12]. Further on, a specialization – the user defined combination of grounder and solver – is described. An example with the grounder *gringo* and the solver *claspD* demonstrates the profit of such a custom combination. This specific combination is explained because it is a use case in the software *ARVis*.

After the description of the most important ASP solvers, a simple table compares the differences between *DLV* and *clingo* (*gringo/clasp*). The last section of this chapter deals with the ASP Wrapper – the embedding of an ASP Solver in external software applications. Such ASP Wrappers are useful to benefit from the powerful features of ASP Solvers in independent programs. Therefore the already existing *DLV Java Wrapper* is introduced [17]. In addition, this section explains how a wrapper could be implemented on your own in case that no wrapper exists for a specific solver or programming language.

With the knowledge about Answer Set Programming, ASP Solvers and ASP Wrappers we can start with the introduction and implementation of the software *ARVis*. This process is described in Chapter 4. Starting with the general background of *ARVis*, the presentation of answer sets and their relationships will be described and illustrated in detail. This section explains what is necessary to put such a visualization process into practice and what details should be considered. For example it is useful to define whether two answer sets correlate or not by a separate ASP program. This way, the user is always able to define the conditions on his own and can do this in a very comfortable way. The next section of this chapter is about the layout and structure of the software. This gives an overview about all existing packages, included libraries and also about the most important classes. After that, short descriptions of the imported libraries are provided to learn about their main features.

Now the implementation of the software gets started by explaining the wizard which is needed for simple program guidance. Short code extracts should demonstrate how the wizard is used. Also, the 5 steps to visualize the relationships between answer sets are illustrated. After that, the connection to the ASP Solver is implemented by using the *DLV Java Wrapper* as well as the self-developed *Gringo-Clasp-Wrapper*. The latter also needs a parser which converts the plain text solutions produced by the ASP Solver in corresponding Java objects for further processing. This is emphasized in the next section by providing the source code of this parser together with a short explanation. Now we are able to calculate and set the relationships between the corresponding answer sets. For that we can use the solutions which we have computed in the previous step and which are now available as simple Java objects. The last section finishes the visualization: a graph is built and the calculated edges and vertices are set.

CHAPTER 2

Answer Set Programming

2.1 Background and application

The idea of Answer Set Programming is to use a program as a formal representation of a certain problem. This means that such a program does not specify how its problem should be solved, but rather describes what has to be achieved and which solutions have to be found. It is a simple declarative logic programming language which includes high-performance abilities to solve computational problems which are specified by a logic program. The solutions of this problem are called answer sets, which the solver has to find and return as output. These results are sets of ground first-order literals.

ASP is one of the most important and popular procedures to solve declarative problems. ASP systems can be used to find solutions for all search problems in non-deterministic polynomial time (NP) or also in NP^{NP}. That is why ASP becomes attractive in the scientific community and it is used in real-world applications with industrial and academic purpose. Some examples for application areas are team-building, reasoning tools in systems biology and synthesis of multiprocessor units. Also, there is a rising interest in artificial intelligence because of the high power of knowledge representing, manipulating and modeling.

The roots of Answer Set Programming can be found in knowledge representation, non-monotonic reasoning, logic programming, Boolean Constraint Solving and also databases. An early version of an ASP solver was *smodels*. Later *DLV* and SAT-based (satisfiability checking) ASP solvers were introduced like *assat* and *cmodels*. Also the conflict-driven ASP solver *clasp* was developed [11, 12].

The structure and components of ASP languages are very similar. The common modeling

components, for example, are disjunction in the heads of the rules, non-monotonic negation, weak/strong constraints and aggregate functions. Most ASP systems support many other interesting and useful features. The language itself is very powerful and expressive, because all properties and rules can be defined as a mathematical formalism. So the problems can be defined in a simple and precise way and it is possible to express all the properties of a finite structure by using a function-free first-order configuration which is decidable in non-deterministic polynomial time with an oracle in NP. Further on, Answer Set Programming is declarative, which means that the order of literals and rules do not matter. ASP is also assigned to Disjunctive Logic Programming (DLP). Therefore, simple disjunction can be used in the heads of the rules and also negation is allowed in the bodies [5, 14, 17].

As mentioned in the introduction, the answer sets computed from the solver are only simple plain text solutions and users might find it difficult to interpret them. Therefore, a few applications were developed, which represent the solutions as visualization, so the user can understand them better. Software applications and tools like *ASPVIZ*[1], *IDPDraw*[2] and also *Kara* [13] are some examples, providing such visualizations of answer sets.

2.2 Syntax and semantics

Programs are defined as collections of clauses of the following form:

$$a_1 \vee ... \vee a_k \leftarrow b_1 \wedge ... \wedge b_m \wedge not\ b_{m+1} \wedge ... \wedge not\ b_n$$

In ASP, strings starting with lower case letters define constants, while strings with uppercase letters are used as variables. This is the same convention as in *Prolog*. Logical variables and constants are called terms. Expressions like $p(x_1, ..., x_n)$ are referred to as atoms, whereas p is a predicate with arity n and the containing arguments are terms. Moreover, literals are defined as positive or negative atoms. An ASP program consists of a finite set of rules.

In written ASP code, disjunctive rules have following expression:

$$a_1 \vee ... \vee a_k :- b_1, ..., b_m, not\ b_{m+1}, ..., not\ b_n$$

$a_1 ... a_k$ and $b_1 ... b_n$ are atoms, whereas $k \geq 0, n \geq m \geq 0$.

The left side of $:-$ is called head, the right side is defined as body. To be more specific, for the rule r, we get $H(r) = a_1, ..., a_k$ as the set of literals in the head, and $B(r) = B^+(r) \cup B^-(r)$ as the set of literals in the body, whereas B^+ is the positive body with $B^+(r) = b_1, ..., a_m$ and B^- is the negative body with $B^-(r) = b_{m+1}, ..., a_n$. The head of a rule must be true whenever

[1] http://www.cs.bath.ac.uk/ occ/aspviz/

[2] http://dtai.cs.kuleuven.be/krr/software/visualisation

all its body literals are true. An atom can only be true if there is at least one rule which is deriving it.

There are a few more definitions describing specific constructs of rules: A rule which contains only one single literal in its head, is called *normal rule*. If a rule does not define any head literals, it is described as *integrity constraint*. This means, the head is assumed to be false. If all literals in the body are true, the entire rule would be false. Therefore the current answer set would be discarded. A rule with an empty body is referred to as a *fact* and in this specific case the sign $:-$ can be omitted. A *fact* is always true.

Usually, all rules in ASP programs have to be safe. That means that each variable in a rule must occur in at least one positive literal in the body of this rule. Further on, a program, a rule or an atom which does not include any variables, is called *ground*.

To understand the semantic background of ASP programs, we have to explain the *Herbrand Universe* and *Herbrand Base* [5, 9, 17]. The *Herbrand Universe*, abbreviated U_P, defines the set of all constants which are contained in a program P. In case no constant is defined in P, the *Herbrand Universe* obtains only one arbitrary constant. The *Herbrand Base*, B_P, is the set of all ground atoms which can be constructed from predicates in P and of all constants which are defined in the *Herbrand Universe*. For every rule r, $ground(r)$ describes the set of rules which are obtained by replacing all occurring variables in r by constants of the *Herbrand Universe*. Moreover, $ground(P)$ is the union of all sets $ground(r)$ for every r in the program P.

We define an interpretation of a program P as a consistent subset of the *Herbrand Base*. The expression *consistent* means that this subset does not include any literal which occurs as its positive and negative form at the same time (for example a and $-a$).

First, let P be a positive program (all rules contain only positive bodies) and I an interpretation: I is closed under P, if $\forall r \in ground(P), H(r) \cap I \neq 0$, whenever $B(r) \subseteq I$. In words, I is closed under P, if for every rule r in $ground(P)$, where the body of the rule is a subset of I, there is at least one element in the head of this rule which occurs in I. An interpretation is an answer set for a positive program P, if it is minimal of all closed interpretations.

To define the answer sets not only for positive but also for general programs, we can use a reduction to positive programs with a stability condition. The *reduct* or *Gelfond-Lifschitz transformation* [5, 9, 17] of a ground program, $ground(P)$, concerning an interpretation I is the positive ground program $ground(P)^I$, which can be obtained by applying the following two rules on $ground(P)$:

- deleting all rules r in $ground(P)$, where the negative body of r contains at least one element of I, or in mathematical term: $B^-(r) \cap I \neq 0$
- deleting all negative bodies from the remaining rules

An answer set for a program P is an interpretation I, if I is an answer set of $ground(P)^I$.

CHAPTER 3

ASP-Solvers

This chapter introduces some popular ASP-Solvers, which we will use and need in the software *ARVis* described in Chapter 4. Also, some custom combinations of grounder and solver will be explained, playing an important role in the *Answer Set Relationship Visualizer* application. Further on, a simple overview represents the differences, advantages and disadvantages of the particular ASP-Solvers. The end of this chapter deals with the embedding of such ASP-Solvers in stand-alone applications.

3.1 DLV

DLV[1] is a state-of-the-art ASP system which is still being developed by Italian researchers. The project started at the end of 1996 as an research project at the Vienna University of Technology which was led by Nicola Leone. It was first released in 1997, and over the years *DLV* has been improved substantially: The language made progress, a lot of extensions were implemented and many optimization techniques were developed, for example database methods for better instantiation and also efficient techniques for model generation and answer set checking. Currently, it supports disjunction, weak-constraints, aggregates and also function symbols. Today, *DLV* is developed in cooperation between the University of Calabria and the Vienna University of Technology. Nowadays, it is widely used all over the world, by researchers as well as in real-world applications. The powerful and efficient *DLV* engine can be seen as competitive to all advanced ASP systems [1].

The following list gives a short overview of some industrial and commercial applications, where *DLV* is a basic component [6, 14]:

[1] http://www.dlvsystem.com

- **Team Buidling in the Gioia-Tauro Seaport:** With the help of *DLV* a system was implemented to automatically and optimally allocate the available personal of the seaport of Gioia Tauro.
- **E-Tourism:** The so-called *IDUM* application is an e-tourism system helping a travel agency to find the best travel routes.
- **Automatic Itinerary Search:** This web application is able to output details about the whole transportation system of the Italian region Calabria, like bus/train departure times, routes, travel time and walking directions.
- **e-Medicine:** In the Italian region Veneto, an automatic system has been developed to classify documents and case histories of clinical diagnoses.
- **Information Integration:** *INFOMIX*, a data integration system, funded by the European Commission, implemented the *DLV System* as computational core. Many web sources and legacy databases have to be integrated for the information system of the University of Rome.
- **CERN:** At the *European Laboratory for Particle Physics*, *DLV* is used as an deductive database application which requires sophisticated knowledge manipulation on huge databases.

DLV is developed using *GNU tools* like *GCC*, *flex* and *bison*. That is why it is portable to Unix platforms and Microsoft Windows. The main component of the system is the *DLV core*. Around this core there are many front-end preprocessors and output filters. *DLV* is executed via the command line and the input data is read from the file/database system. Also, an *Intelligent Grounding Module* is implemented. Its main task is to generate a subset of the grounded input. This subset is usually much smaller than the original but still has the same answer sets. This modified ground program is then processed by the *Model Generator*, which computes an answer set and proofs with the help of the *Model Checker* whether it is valid or should be discarded. This is done with every single answer set until a predefined maximum number has been generated or no more answer sets are available. Finally the result is filtered and printed to the output [6, 14].

As mentioned above, *DLV* is very powerful and has many language enhancements. To go in more detail: *DLV* has capabilities for advanced knowledge modeling, like an efficient formalism for knowledge representation and reasoning. This high-expressive language construct includes aggregates, strong/weak constraints, lists, sets, functions and more. Also inheritance and queries, built-in predicates and also integer arithmetics are available, of course. The order of rules is not relevant, so *DLV* is full declarative. Many front-ends exist supporting special applications in the fields of artificial intelligence and information extraction [4, 6, 14].

Further on, the implementation of the *DLV engine* is very solid, so many algorithms and techniques for database and search optimization are available to increase the performance. Such improvements are indexing and join ordering methods and also heuristics and back-jumping

techniques. An advantage of the high-performance of *DLV* is that it can easily handle complex problems and also is able to manage applications with enormous use of data [14].

The last important area is the interoperability of *DLV* to interact with external applications and systems: many abilities exist to connect to relational database management systems over a *ODBC* interface or to call external *C++* functions within the *DLV programs*. Also a mechanism for running *DLV* from Java programs is available, the *DLV Java Wrapper*[2]. This *Java Wrapper* is introduced in Section 3.5.

3.2 Clingo (gringo + clasp)

The university of Potsdam develops Answer Set Programming tools such as *gringo*, *clasp* and *clingo*[3].

Gringo is a grounder tool and is able to translate logic programs with first-order variables into equivalent propositional logic programs. This is necessary because answer set solvers are based on variable-free (ground) programs. After that, the application *clasp* can be used to generate the corresponding answer sets. *Clasp* is an answer set solver, which provides powerful modeling capacities and state-of-the-art techniques from Boolean constraint solving. For satisfiability checking (SAT) *clasp* uses an algorithm based on *conflict-driven nogood learning*. The last tool, *clingo*, combines both features in one single application [10, 12].

All three software applications are written in *C++* and are published under the *General Public License* (*GNU*). Precompiled binaries and source packages are available for Linux and Windows.

By using *gringo* and *clasp*, the output of *gringo* is usually piped to *clasp*. Therefore, the following two commands are the same:

```
gringo [ options | files ] | clasp [ options | number ]
clingo [ options | files | number ]
```

The **options** argument: There are a lot of options available which can be used to set extra functionality or to define some preferences and limits. The **files** argument: Several files can be added here, which should be used as input. The **number** argument: it defines the maximum number of answer sets which should be computed.

[2] http://www.dlvsystem.com/dlvsystem/index.php/DLV_WRAPPER

[3] http://potassco.sourceforge.net/

A simple *gringo/clasp* invocation could be as follows:

```
gringo input_A.lp input_B.lp | clasp
```

where `input_A.lp` and `input_B.lp` are the source files.

Like *DLV*, *gringo/clasp* and *clingo* have many powerful features: on the one hand, they support the characteristic ASP main features like integrity constraints, classical negation and disjunction. On the other hand they provide built-in arithmetic functions, built-in comparison predicates, assignments, intervals, conditions, pooling, aggregates, optimization, meta-statements and integrated scripting language [10].

To explain some of these expressive features: *Intervals* can be used in the form `i..j`, where `i` and `j` are integers. Then, such an interval defines each integer `k` which is between `i` and `j`. All intervals are expanded by the grounding process. For example: `num(1..3).` is the same as `num(1). num(2). num(3).`

Conditions are defined by the symbol `:` and are helpful to encode a lot of ground atoms that are connected by conjunctions and disjunctions. The advantage of this construction is the more efficient and shortened notation. A short example:

```
pc(1). pc(2). pc(3). pc(4). pc(5)
assign(X) : pc(X).
```

is the same as

```
pc(1). pc(2). pc(3). pc(4). pc(5)
assign(1) | assign(2) | assign(3) | assign(4) | assign(5).
```

Another interesting feature is *pooling*. With the symbol `;` it is possible to specify alternative terms for arguments inside an atom. Therefore, `p(...,X;Y,...)` in the body of a rule is the same as `p(...,X,...), p(...,Y,...)`, whereas such an expression inside the head of a rule causes to expand to multiple rules.

Aggregates can be used on a multiset of weighted literals that results to a specific value. *Gringo* provides `#sum`, `#min`, `#max` to compute the sum, minimum and maximum of weights for example.

With *optimization* statements answer sets can be rated. This is like the *weak constraint aggregate* in *DLV*. With *meta-statements* it is easy to write comments, hide predicates, use constants, declare domains and do a lot more. The last relevant feature and ability is to use the integrated scripting language, so-called *lua*, inside a program which provides extended features like inserting answer sets into a database.

3.3 Specialization: User defined combination of grounder and solver

In some cases it is useful to define a custom combination of grounder and solver. Of course, they have to be compatible to each other, so in our case we use the tools from Potassco[4], *gringo* and *claspD*. *Gringo* is the grounder, already mentioned in Section 3.2, *claspD* is an extended version of the *clasp* solver, which is able to solve disjunctive logic programs.

Existing ASP systems do not provide any complex optimization capacities, but Potassco is offering a collection of meta-encodings which support such optimizations and preferences by using simple meta-programming [3]. Therefore, *gringo* has to be called with the source files and the *–reify* option enabled to get a reified output from *gringo*. That means that the result is an output ground program in form of facts. This output can be used as input for another *gringo* invocation, and in this case we only have to specify the meta-encodings. Further on, this resulting output has to be transmitted to the *claspD* solver, which finally computes the desired answer sets.

To give a concrete example, we assume to search for cardinality-minimal answer sets. To sum it up, we need a *gringo-gringo-claspD* invocation as follows:

```
gringo –reify source.lp | gringo - encodings/{meta.lp,metaD.lp,
metaO.lp} <(echo "optimize(1,1,card).")  | claspD 0
```

whereas `source.lp` is the relevant source file and `meta.lp`, `metaD.lp` and `metaO.lp` are the meta-encodings which are located in the encodings folder. The component `<(echo "optimize(1,1,card).")` only inserts an additional fact for the optimization, meaning that all literals of priority level and weight 1 are an issue of cardinality minimization [3].

3.4 Differences

Now we want to give an overview about the differences between the two ASP solvers *DLV*[5] and *clingo*[6] (or respectively *gringo*[7]/*clasp*[8]). Both have powerful engines with a lot of features, but *DLV* offers better interoperability with external applications like the *Java Wrapper*, *ODBC* interface and the ability to call external *C++* functions. It also supports queries which means that *DLV* is able to handle incoming queries about existing facts. *Gringo/clasp* does not have as much compatibility features as *DLV* but also provides an integrated scripting language with database support. In contrast to this restriction, *gringo/clasp* has more language constructs with the ability

[4]http://potassco.sourceforge.net/
[5]Version from December 21st, 2011
[6]Version 3.0.4
[7]Version 3.0.4
[8]Version 2.0.6

for conditions and pooling and it has better aggregate functions and meta-statements than *DLV*. For example with *gringo/clasp* it is possible to define lower and upper bounds by using aggregate functions.

	DLV	gringo/clasp
strong integrity constraints	available	available
weak integrity constraints (optimization)	available	available
classical negation	available	available
disjunction	available	restricted (fully available only in claspD)
built-in arithmetic functions	available	available
built-in comparison predicates	available	available
assignments	available	available
intervals	available	available
conditions		available
pooling		available
aggregates	basic	extended
queries	available	
meta-statements	less	a lot
interoperability	Java Wrapper, ODBC interface, external C++ function calls	integrated scripting language with database support

Table 3.1: Differences between DLV and gringo/clasp

3.5 ASP Wrapper: Embedding ASP-Solver in independent external applications

Most ASP solvers do not provide any libraries or interfaces to communicate and interact with the solver from external applications. That means, it is often not possible to integrate the solver in an independent software development language like C++ or Java, in order to use it for knowledge representation and reasoning or solving disjunctive logic programs.

One exception is *DLV*, because for this ASP solver a Java library has been developed, which can be used to "wrap" the *DLV System* in a Java application. The so-called *DLV Wrapper*[9] is a simple object-oriented library, implemented in Java. Therefore, the *DLV Java Wrapper* can be seen as interface between the external application and the *DLV System*. This way, it is quite easy to run logic programs and to process the resulting output with normal Java code. The *DLV Wrapper* is divided in many classes, which can be used to have full control over the *DLV* execution. For example, there are classes like `Literal`, `Predicate`, `Model` and `Program`, which provide helpful functions to run the program and read and process the output. In fact, the result is not only a simple string representation of answer sets. Furthermore each answer set is returned as `Model` object which can be handled and passed through by implemented methods to get all its predicates and literals which are also represented by `Predicate` and `Literal` objects. That is why the complete process of running *DLV* in an external application is quite comfortable. Of course, it is also possible to set invocation parameters and to fully control the *DLV* execution, for example to check the exit status or destroy/stop the running *DLV* process [17].

To integrate an ASP solver in external applications where no libraries and no interfaces exist, for example *gringo/clasp*, it is necessary to develop such an interface on your own. First, the relevant ASP solver or respectively the grounder has to be executed with all input files and arguments. Therefore, it could be helpful to implement such features by using and running a separate thread, for better usability and avoiding to freeze the user interface by executing long and complex logical programs. By using a separate solver, the output (the grounded program) has now to be piped to the corresponding ASP solver applying the same procedures as above. This being done, the resulting output has to be parsed to process the content (literals, predicates and terms) of the answer sets in the further application context. More details about the implementation are described in Section 4.5 when we need to build such a component for our *Answer Set Relationship Visualizer*.

[9] http://www.dlvsystem.com/dlvsystem/index.php/DLV_WRAPPER

CHAPTER 4

Software “ARVis”

4.1 Background: The presentation of answer sets and their relationships

The main topic of this bachelor’s thesis is the visualization of relationships between answer sets. But how can this be put into practice? How is it possible to indicate that two answer sets have common properties or interrelate?

The basic idea is to use a graph, where a vertex represents a single answer set and the relationship between two answer sets is illustrated as an edge. But to develop a software that is solely focused on this problem reveals many other questions: How can the relationships be set? How is it possible to define them as a result of simple rules?

A method is required which sets the relationships between answer sets automatically. This can be best realized with another answer set program, where the result defines those connections. This being done, setting relationships where answer sets have a common property or interrelate becomes possible.

The challenge now is to define a simple software which supports all these features. It has to require two ASP files from the user – one for the main task and one for the visualization. Afterwards the corresponding graph could be drawn. But there is still another problem: The connection between the main program and the visualization program. The latter requires the result of the first program. That means, the result has to be parsed and processed, so that it is compatible with the visualization program. To keep it simple: The result of the main program – all answer sets together which are needed as input for the second program – has to be converted into the corresponding ASP code. This code is only a simple list of ASP conforming “facts”.

Each of these facts consists of a single literal or respectively of a predicate. The predicate's name could be any word, but for better comprehension, we use "as", meaning answer set. The arguments of this predicate can be as following: an identity of the related answer set, the name of the corresponding predicate in the model of this answer set and the last arguments are taken over from the related predicate. This way the relationship between the fact and the answer set is maintained.

The following pattern is advisable:

```
as(answerset_id, predicate_name, arg1, arg2, arg3, ...)
```

An example: Suppose that we get the following output from a simple basic answer set program with 2 answer sets. The semantics of the predicates "in" and "out" are not relevant for the conversion.

```
1: in(1, 2) in(1,3) in(1,4) out(1,5)
2: in(2, 2) in(2,3) in(2,4) out(4,5)
```

This output would be converted to:

```
as(1, in, 1, 2).
as(1, in, 1, 3).
as(1, in, 1, 4).
as(1, out, 1, 5).

as(2, in, 2, 2).
as(2, in, 2, 3).
as(2, in, 2, 4).
as(2, out, 2, 5).
```

In fact, we get an ASP code, which can be used in another context again – in our case the visualization program to calculate the relations.

Executing the two relevant files – the main ASP file and the corresponding visualization ASP file – outputs a result, where the answer sets define the edges in a graph. The only step left to do is to parse the result of the ASP-Solver and generate graph components which can be delivered to the graph visualization. Furthermore, the visualization program possibly outputs multiple answer sets. This way, it is simple to generate several different models. Each of these models represents a graph of their own and therefore their own visualization.

Now we managed to get the whole visualization process done. Figure 4.1 demonstrates the just explained idea.

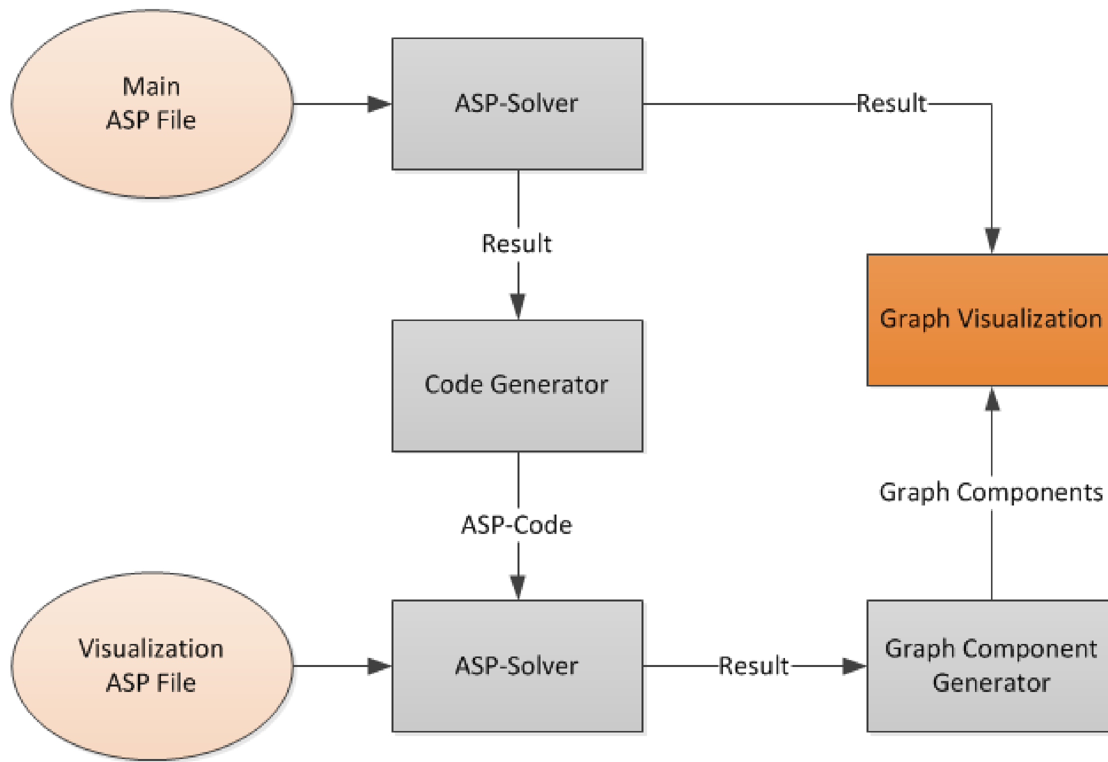

Figure 4.1: Process of visualization

4.2 Layout and structure of the software

The program *ARVis* is divided into 5 main packages. Each of them has a separate and independent main task. Figure 4.2 shows the general structure, the used libraries and the hierarchy of the packages.

Package *ASP*:
The main function of this package is the representation of an answer set and it includes classes for data management like internal data structures for the complete process of the software. So, classes like `AnswerSet`, `Literal` and `Result` can be found there. Furthermore help functions are implemented in order to convert answer sets into ASP code and to process them in another context again. In our case, this other context is the evaluation of the relationship between the answer sets. A sub package of the ASP package, called *wrapper*, is responsible for the connection between the *ARVis* software and the corresponding ASP-Solver. Therefore, this is one of the most important parts of the software and we will examine it in more detail later on.

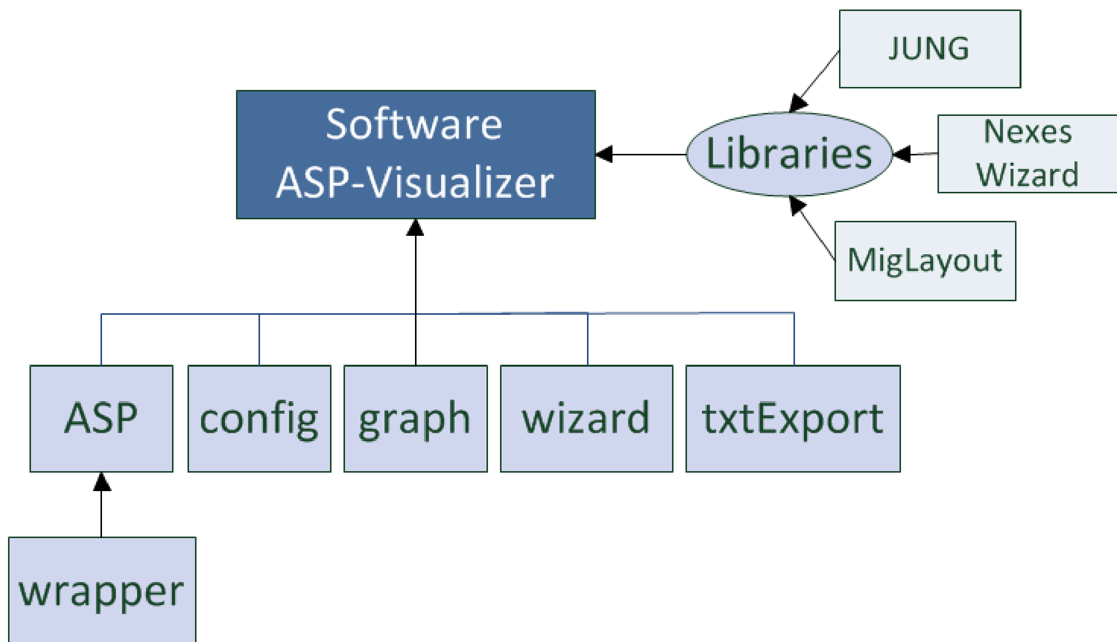

Figure 4.2: Structure of ARVis

Package *config*:
This module is essential for the configuration and the storage of user defined preferences. All data is stored in a simple property file, which can also be edited manually because of the fact that it is saved as plain text. Additionally, the `Memory` class is responsible for the temporary storage of all data which is required during the steps in the wizard panel.

Package *graph*:
To visualize the relationship and the connections between two or more answer sets, the classes of this package are used. Some classes are only important for data storage and data structure, for example `Edges`, `Edge` and `Vertices`. These are relevant for the class `VisualizationResult`, which is the representation of the final result. It contains all relationships of the answer sets which have to be visualized in a graph - consisting of edges and vertices and the optional highlighted ones. To generate a user-friendly graph, the class `GraphVisualization` is needed. It uses the `VisualizationResult` class to get the necessary data and draw a graph in the user interface. This graph is not only a static image, it is interactive and the user can move each single vertex and also resize the whole graph. Each vertex is clickable, to select one or more answer sets, whose content is displayed in the text box right of the graph. All this graphical features are supported by the *JUNG* library.

Package *wizard*:
All classes of this package are responsible for the general user interface. The complete process of the *Answer Set Relationship Visualizer* is done in a wizard, i.e. the user is guided through the

program step by step. The great advantage of this method is the better management and control for the user. All panels which are used in this wizard are found in this package. Each step of the wizard is implemented in a separate class as a panel. Also a so-called `ProgressViewer` is developed for better usability. It indicates the current status and displays which steps are done and which have to be passed through.

Package *txtExport*:
This package contains only one single class, whose main function is to generate a text report of the final outcome.

As mentioned above, the software is grouped in modules or rather in packages. To explain all features and methods of each class in every package would go beyond the scope of this Bachelor's thesis, so we would like to explain only the most important and interesting components and classes in this project. For better comprehension a source code of *ARVis* is included in some cases.

4.3 Interfaces & libraries

Some libraries were used to add extra functionality and to build on existing java classes and interfaces. The following libraries are used:

DLV-Wrapper:
This library provides a simple java interface to the DLV System. This means that the DLV-Wrapper allows to embed the disjunctive logic programs written in DLV into a java program. With the powerful methods of this wrapper, the DLV can be controlled and accessed completely via simple Java code.

JUNG 2.0.1: [2]
JUNG is the *Java Universal Network/Graph Framework* and is an open-source Java library which supports the modeling, visualization and analysis of data in a graph. It also uses other existing third-party java libraries. *JUNG* includes many algorithms from graph theory and is able to represent a lot of various graphs like directed and undirected graphs, hypergraphs and multi-modal graphs. The vertices and edges can be annotated with metadata and descriptions and the layout of the whole graph can be easily adapted to a customized one. *ARVis* requires some parts of these powerful features, whose implementation is mentioned in Section 4.8.

Nexes Wizard: [7]
This is a free Java library which supports to create a simple wizard. It allows to create typical `JPanel` objects , which can be added to an instance of the wizard class. The wizard automatically includes the buttons *Next* and *Back* to each window and is responsible for displaying the corresponding next or previous panel. The detailed implementation and use in *ARVis* is described in the Section 4.4.

MigLayout:
MigLayout is also a free and open software. It provides a Java layout manager which simplifies creating layouts and user interfaces.

4.4 Use of the wizard for program guidance

A wizard is implemented to keep the user interface simple. This way, the user can be lead from one step to another. Each step is based on the result and input from the previous one. That is why a wizard is the best and efficient possibility to design the user interface. The *Nexes Wizard* is a Java library that fulfills these requirements. It provides simple classes to create a wizard.

The basic idea is to implement descriptor classes which are inherited from the abstract class `WizardPanelDescriptor` provided by *Nexes Wizard*. Such a descriptor class has the following structure:

```
public class WizardPanel01_Descriptor extends WizardPanelDescriptor {

  public static final String IDENTIFIER = "ExecuteDLV";
  private WizardPanel01 panel;

  public WizardPanel01_Descriptor() {
    panel = new WizardPanel01();

    setPanelDescriptorIdentifier(IDENTIFIER);
    setPanelComponent(panel);
  }

  public Object getNextPanelDescriptor() {
    return WizardPanel02_Descriptor.IDENTIFIER;
  }

  public Object getBackPanelDescriptor() {
    return null;
  }

  public void aboutToHidePanel() {
    ...
  }

  public void aboutToDisplayPanel() {
    ...
  }
  ...
}
```

The constructor of this class creates a new object `WizardPanel01`, which is a subtype of the `JPanel` class. Therefore, every `JPanel` can be added to the wizard. This new object and

an identifier are assigned to the `WizardPanelDescriptor`. For every new panel that has to be inside the wizard, a new class that extends from `WizardPanelDescriptor` must be implemented.

Furthermore, some methods exist and must be overridden in order to take control of the wizard. Such methods are `getNextPanelDescriptor()` to return the identifier of the next panel, `getBackPanelDescriptor()` to set the previous one, `aboutToDisplayPanel()` to define some code which has to be executed when the panel is displayed and the last important method `aboutToHidePanel()`, which is called before the next panel is shown.

The following source code gains insight, how the wizard is configured:

```
JFrame frame = new JFrame();
frame.setTitle("ARVis");

...

Wizard wizard = new Wizard(frame);

...

descriptor1 = new WizardPanel01_Descriptor();
wizard.registerWizardPanel(WizardPanel01_Descriptor.IDENTIFIER, descriptor1);

WizardPanelDescriptor descriptor2 = new WizardPanel02_Descriptor();
wizard.registerWizardPanel(WizardPanel02_Descriptor.IDENTIFIER, descriptor2);

...

wizard.setCurrentPanel(WizardPanel01_Descriptor.IDENTIFIER);
int ret = wizard.showModalDialog();
```

First, a new `Wizard` object has to be created and then the `WizardPanelDescriptor` objects can be added by using the `registerWizardPanel` method. Afterwards, the current panel of the wizard can be set and the last thing to do is to call `showModalDialog()`, which displays the wizard.

For the *Answer Set Relationship Visualizer* we need 5 `JPanels` because 5 steps are necessary to do the process of visualizing. Figure 4.3 shows the 5 steps in detail which are represented by a gray rectangle. In order to implement a common wizard, the user must have the opportunity to go backwards to change inputs.

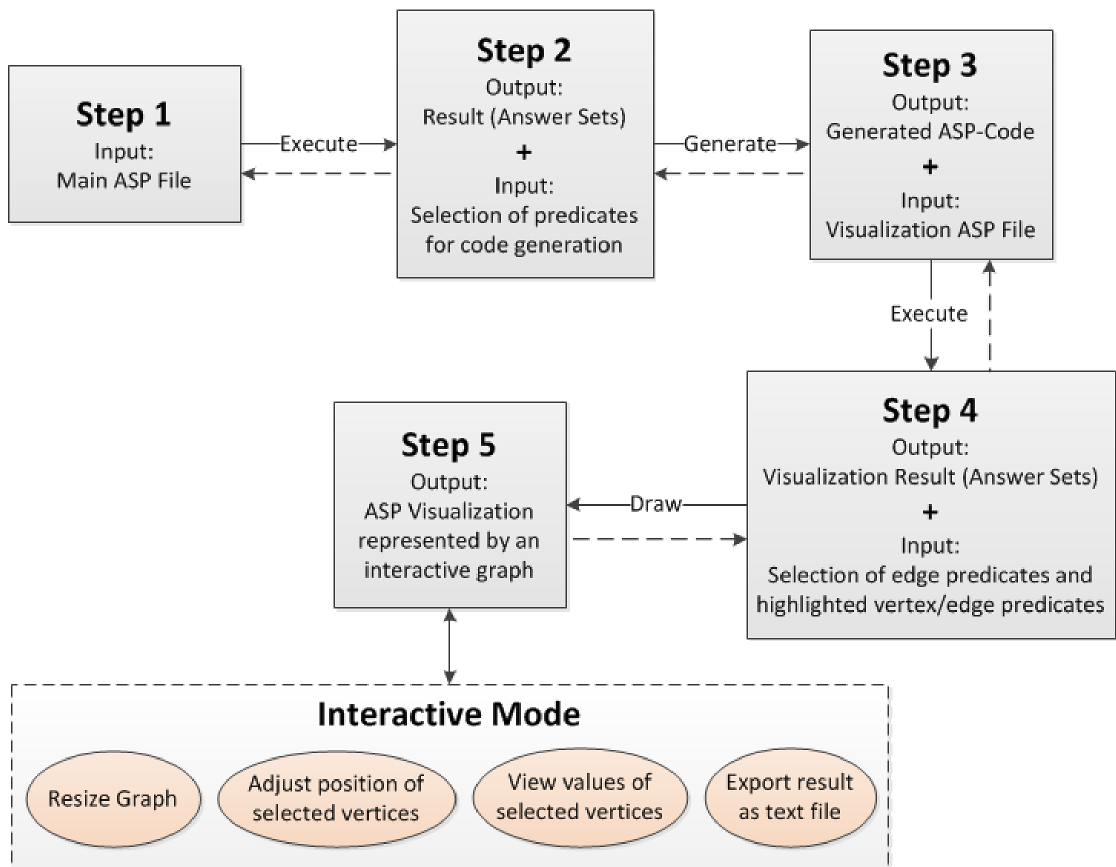

Figure 4.3: 5 steps to visualize the relationship between answer sets

4.5 Connection to ASP-Solver

In order to execute the relevant ASP code and process the result in further steps, a connection between *ARVis* and the corresponding ASP-Solver is essential. Therefore, an abstract class is advisable which provides the necessary interface to interact with the ASP-Solver. The implementation for a specific ASP-Solver, like *DLV* and *Clingo*, is then introduced as a sub class to provide a common interface.

The following source code shows the abstract class `ASPWrapper`:

```
1 public abstract class ASPWrapper {
2
3   protected List<String> files;
4   protected String code;
5   protected int maxAnswerSets;
6   protected boolean executionStopped;
7   ...
```

```
8
9    public static ASPWrapper createWrapper(boolean firstExecution) {
10     Config config = Config.getInstance();
11     String solver = config.getProperty(Config.SOLVER);
12     ASP_Solver asp_solver = ASP_Solver.valueOf(solver);
13
14     if(asp_solver == ASP_Solver.Clingo) {
15       return new ClingoWrapper();
16     }
17     else if(asp_solver == ASP_Solver.Other) {
18       return new GringoClaspWrapper(firstExecution);
19     }
20     else {
21       return new DLVWrapper();
22     }
23   }
24
25   ...
26
27   public void setFiles(List<String> files) {
28     this.files = files;
29   }
30
31   public void setCode(String code) {
32     this.code = code;
33   }
34
35   ...
36
37   public synchronized void stopExecution() {
38     this.executionStopped = true;
39   }
40
41   public abstract Result execute() throws ExecutionException;
42 }
```

This `ASPWrapper` provides simple methods like `setFiles` to set a list of files that should be executed, or `setCode` to deliver a code as string. The static method `createWrapper` returns a specific wrapper whereas the type depends on the current configuration. To create a new specific wrapper class, the `ASPWrapper` has to be inherited and the only method `execute()` has to be implemented. This method is responsible for the complete execution of the ASP code and has to return the answer sets in a new object called `Result`. Additionally, the method `stopExecution()` can be overridden providing to stop the execution process.

Three specific `ASPWrapper` classes were established to support *DLV*, *Clingo* and a customized combination of *Gringo & Clasp*.

The `DLVWrapper` implementation uses the already existing *DLV Java Wrapper* which is the official library of the *DLV System*.

Here is an extract of the `DLVWrapper` class:

```
public class DLVWrapper extends ASPWrapper {
  ...

  @Override
  public Result execute() throws ExecutionException, ParserException,
      InterruptedException {
    Program p = new Program();

    for(String file : this.files) {
      p.addProgramFile(file);
    }

    p.addString(this.code);

    ...

    DlvHandler dlv = new DlvHandler(path); // set path DLV executable
    dlv.setProgram(p); // sets input (contained in a Program object)

    ...

    List<AnswerSet> answerSets = new ArrayList<AnswerSet>();

    ...

    dlv.run(DlvHandler.ASYNCHRONOUS); // run a DLV process and set the output
        handling method.

    while(dlv.getStatus() != DlvHandler.FINISHED) {
      Thread.sleep(100);

      if(this.executionStopped == true) {
          dlv.kill();
          throw new InterruptedException();
      }
    }

    while(dlv.hasMoreModels()) { // while DLV outputs a new model

      Model m = dlv.nextModel();

      ...

      while(m.hasMorePredicates()) {
      {
        Predicate pr = m.nextPredicate();

        while(pr.hasMoreLiterals()) {
          Literal lit = pr.nextLiteral();
          String predicateName = "";

          if(lit.isPositive()) {
```

```
51              predicateName = lit.name();
52            }
53            else {
54              predicateName = "-" + pr.name();
55            }
56
57            asp.Literal literal = new asp.Literal(predicateName);
58
59            for(int i = 0; i < lit.arity(); i++) {
60              literal.addTerm(lit.getTermAt(i));
61            }
62
63            answerSet.addLiteral(literal);
64          }
65        }
66        answerSets.add(answerSet);
67      }
68
69      ...
70
71      return new Result(answerSets);
72    }
73  }
```

In fact, the only method that has to be overridden is the `execute()` method:
First a new DLV program is created and all relevant ASP code and ASP files are added. Then a new object of `DlvHandler` is instantiated, whose only argument is the path to the DLV executable. After the DLV program was set in the `DlvHandler` we can begin with the execution by calling the run method. With the help of the methods `nextModel()`, `nextPredicate()`, `nextLiteral()` and `getTermAt()` we can easily navigate through the result and receive all relevant data that we need for our own data object of `AnswerSet`. The return value of the `execute()` method is a new object of type `Result` which contains all `AnswerSet` objects.

Now we can go on to the more complex `GringoClaspWrapper`. In this case, no library exists that provides an java interface to the *Glingo* and *Clasp* executables. That's why the execution of the *Glingo* and *Clasp* program has to be done manually with the needed ASP code as input stream. Furthermore, a parser has to be introduced which converts the output of the executable into a data object of type `Result`, we can deal with in further context. The implementation of the parser will be discussed in Section 4.6.

The user of the software is able to define a custom combination of Glingo and Clasp in the configuration. This string can consists of several commands separated by the pipe character |. An example for such a string is:
`gringo -reify | gringo - encodings/meta.lp encodings/metaD.lp encodings/metaO.lp input.lp | claspD 0`
where the ASP code would be transmitted by an input stream to the command `gringo -reify`.

The main task is to set the relevant ASP code as input stream for the first command and the output as input for the following command. The output of the last command is the result containing the answer sets as plain text.

The following source code is an extract of the real `GringoClaspWrapper`:

```
public class GringoClaspWrapper extends ASPWrapper {
  ...

  public Result execute() throws ExecutionException, ParserException,
      InterruptedException {
    private Process process;
    ...

    // read the input of the source files
    for(String file : this.files) {
      ...
      Scanner scanner = new Scanner(new FileReader(file));

      while(scanner.hasNextLine()) {
        input += scanner.nextLine() + "\n";
      }
      ...
    }

    input += this.code;

    String[] exec_parts = executeString.split("\\|");

    try {
      for(int i = 0; i < exec_parts.length; i++) {
        File workingDir = null;

        if(execPath.isEmpty() == false) {
          workingDir = new File(execPath);
        }

        String exec = exec_parts[i].trim();
        String[] cmdParts = prepareExec(exec, execPath);

        // execute the command
        process = Runtime.getRuntime().exec(cmdParts, null, workingDir);

        if(this.executionStopped == true) {
          process.destroy();
          throw new InterruptedException();
        }

        // set the input for the executable
        PrintWriter writer = new PrintWriter(process.getOutputStream());
        writer.print(input);
        writer.close();

```

```
47        result = "";
48
49        // get the output from the executable
50        Scanner scanner = new Scanner(process.getInputStream());
51        while(scanner.hasNextLine()) {
52          String line = scanner.nextLine();
53          result += line + "\n";
54        }
55
56        // get the error message from the executable
57        scanner = new Scanner(process.getErrorStream());
58        while(scanner.hasNextLine()) {
59          String line = scanner.nextLine();
60          error += line + "\n";
61        }
62
63        // set the result as input for the next command
64        input = result;
65
66        if(this.executionStopped == true) {
67          throw new InterruptedException();
68        }
69      }
70    }
71    catch(InterruptedException e) {
72      throw new InterruptedException();
73    }
74    catch(Exception e) {
75      throw new ExecutionException(e.getMessage());
76    }
77
78    ClaspResultParser parser = new ClaspResultParser(result, error);
79    return new Result(parser.getAnswerSets());
80  }
81
82  private String prepareExec(String exec, String workingDir) {
83    if(workingDir == null) {
84      workingDir = "";
85    }
86
87    String[] cmdParts;
88    ...
89    boolean inQuotes = false;
90    String buffer = "";
91    List<String> cmdList = new ArrayList<String>();
92
93    // parse the command string and split it to use it for the exec method (
        string array containing file, arg1, arg2 ...)
94    for(int i = 0; i < exec.length(); i++) {
95      // double quotes can be used to use spaces in path and arguments; the
          boolean inQuotes has to be true when we are currently parsing
          between two double quotes
96      if(exec.charAt(i) == '"') {
```

```
97          inQuotes = !inQuotes;
98          continue;
99        }
100
101       // if the current character is a space and this character is NOT
              between two double quotes, then add the current string in the
              buffer as a new element to cmdList and clear the buffer
102       if(exec.charAt(i) == ' ' && inQuotes == false) {
103         cmdList.add(buffer);
104         buffer = "";
105         continue;
106       }
107
108       //add the current character to the string buffer
109       buffer += exec.charAt(i);
110     }
111
112     //finally add the remaining string to cmdList
113     cmdList.add(buffer);
114
115     cmdParts = cmdList.toArray(new String[0]);
116
117     if(new File(cmdParts[0]).isAbsolute() == false) {
118       cmdParts[0] = workingDir + "/" + cmdParts[0];
119     }
120
121     if(new File(cmdParts[0]).isFile() == false) {
122       throw new ExecutionException("The file \"" + cmdParts[0] + "\" is
              incorrect.\nPlease enter a valid file in Menu -> Start ->
              Configuration");
123     }
124
125     return cmdParts;
126   }
127
128   @Override
129   public synchronized void stopExecution() {
130     executionStopped = true;
131
132     if(process != null) {
133       process.destroy();
134     }
135   }
136 }
```

First, the code contained in the ASP files is read and then the additional code that was set in the variable `code` is added. This optional data string containing in the variable `code` is only set in the second ASP execution (visualization program), when we need the generated ASP code from the first execution (main program). The merged code has to be set as input stream for the first command. To get the several commands, the complete command string has to be split by the pipe character. In the implemented loop each command is executed with the corresponding data

as input stream and afterwards the resulting output is set as input for the next command. After this loop we get the desired output in the variable `result` that contains the answer sets which we need for further visualization. But this result is only a plain text which has to be parsed with the help of the developed `ClaspResultParser` to receive a list of `AnswerSet` data objects.

Additional information about the working directory:
When a command is executed, the environment for this executable must be set to the user-defined working directory. Also, the command itself has to be prepared which is done by the method `prepareExec()`. This method only inserts the working directory at the beginning, if the command does not contain an absolute file path. This preparation is done because then the user does not have to include the absolute path in every command and in every argument – it simplifies the user-defined command string very much.

4.6 Parser: The processing of the result data

As mentioned above, for *Clingo*, *Clasp* and *ClaspD* we need a parser which does the processing of the output text. The output of the three ASP-Solvers is very similar because they descend from the same solver family, but they differ in some aspects like various offsets of the relevant data or different number of line breaks. So, the best thing to do is to combine the functionality of parsing the output of all three solvers in one single class. This implementation is then able to handle all outputs, independent of the specific type. An advantage is, that other similar ASP-Solvers can be used as long as they have one of the three output formats.

For the *DLV System* we do not need anything like that, because the *DLV Java Wrapper* library includes all these features.

To demonstrate how the parser works, here is the simplified version of the source code of the `getAnswerSets` method in the the class `ClaspResultParser`:

```
1  public List<AnswerSet> getAnswerSets() throws ParserException
2  {
3    List<AnswerSet> answerSets = new ArrayList<AnswerSet>();
4
5    // split the output into single lines
6    String[] lines = result.split("\\n");
7    int offset = -1;
8
9    // determine the offset of the real data
10   for(int i = 0; i < lines.length; i++) {
11     if(lines[i].contains("Answer") == true) {
12       offset = i;
13       break;
14     }
15   }
16   ...
```

```
17
18    int id = 1;
19
20    // pass through every line
21    for(int i = offset; i < lines.length; i++) {
22      // break if a special keyword occurs
23      if(lines[i].equals("SATISFIABLE") || lines[i].equals("UNSATISFIABLE,") ||
            lines[i].equals("UNKNOW") || lines[i].isEmpty()) {
24        break;
25      }
26
27      // only each second line has relevant data
28      if((i - offset) % 2 == 0) {
29        continue;
30      }
31
32      // split a line into literals
33      String[] lits = lines[i].split(" ");
34      AnswerSet answerSet = new AnswerSet(id);
35
36      // pass through the literals
37      for(int j = 0; j < lits.length; j++) {
38        // create a new literal object and add it to the current answer set
39        Literal literal = new Literal(getPredicate(lits[j]), getTerms(lits[j]))
              ;
40        answerSet.addLiteral(literal);
41      }
42
43      answerSets.add(answerSet);
44      id++;
45    }
46
47    return answerSets;
48  }
```

The first lines of this source code determine the offset of the real relevant data. The beginning of the output is some text which is irrelevant for parsing, so we are going to skip it until we reach the first answer set solution. Then we pass through the lines and split every line containing an answer set into its literals. It is important that only every second line includes a model, all others lines are skipped. When we now obtain the single literal strings, we use the two methods `getPredicate()` and `getTerms()` to receive the predicate as well as the list of terms containing in the string representation of the literal. These two help methods extract the data by splitting the string, similar to the method above. The name of the predicate and the list of terms are set in a new `Literal` object, which is added to the current answer set. Each of these answer sets is an element of an `ArrayList`. After parsing the mentioned `ArrayList` object is returned which contains all concerning models.

4.7 Calculation and setting of relationships

When we run the the main program, generating the corresponding ASP code from its result and adding this code to the visualization program, we get an `Result` object as output. The content of this data object is a list of all models where each of them represents an independent solution for visualization. But these models only contain all data in the form of ASP related *Literals*, so we still need to calculate and generate `Edge` objects for the relationships between the answer sets of the main program. This `Edge` objects are necessary to draw a graph.

To store these `Edge` objects in the same `Result` object, we have to create a new class called `VisualizationResult` which extends the `Result` class to provide additional features. Therefore, a new method `generateEdges(List<String> predicates, List<String> predicatesHighlighted)` is implemented, which is responsible for the generation of these `Edge` objects. The first argument is a list of all predicates representing an edge, the latter list defines which edges should be highlighted with a special color.

The following code presents the functionality of this extended class:

```
public class VisualizationResult extends Result{
  private List<Edges> edges;
  private List<Vertices> vertices;

  ...

  public void generateEdges(List<String> predicates, List<String>
      predicatesHighlighted) {

    Config config = Config.getInstance();
    boolean generateOneGraph = config.getBooleanProperty(Config.
        GENERATE_ONE_GRAPH);

    int answerSetsNumber = Memory.getInstance().getResult().getAnswerSets().
        size();

    // pass through all existing answer sets
    for(AnswerSet answerSet : this.getAnswerSets()) {
      Edges e = new Edges();
      Vertices v = new Vertices();

      // pass through all literals in this answer set
      for(Literal lit : answerSet.getLiterals()) {

        ...

        // check whether the predicate represents an highlighted edge or
            vertex
        if(predicatesHighlighted.contains(lit.getPredicate() + "/" + lit.
            getTerms().size())) {
          // check whether the predicate represents an edge (binary) or a
```

```
                vertex (unary)
27            if(lit.getTerms().size() == 2) {
28              int from  = Integer.parseInt(lit.getTerms().get(0));
29              int to = Integer.parseInt(lit.getTerms().get(1));
30
31              if(from >= 1 && from <= answerSetsNumber && to >= 1 && to <=
                    answerSetsNumber) {
32                // add the edge to the Edges object with highlighted set true
33                e.add(from, to, true);
34              }
35            }
36            else if(lit.getTerms().size() == 1) {
37              int vertex = Integer.parseInt(lit.getTerms().get(0));
38
39              if(vertex >= 1 && vertex <= answerSetsNumber) {
40                // add the vertex to the Vertices data object
41                v.add(vertex);
42              }
43            }
44          }
45
46          // check whether the predicate represents an edge
47          if(predicates.contains(lit.getPredicate() + " / " + lit.getTerms().
                size())) {
48            if(lit.getTerms().size() == 2) {
49              int from  = Integer.parseInt(lit.getTerms().get(0));
50              int to = Integer.parseInt(lit.getTerms().get(1));
51
52              if(from >= 1 && from <= answerSetsNumber && to >= 1 && to <=
                    answerSetsNumber) {
53                // add the edge to the Edges object with highlighted set false
54                e.add(from, to, false);
55              }
56            }
57          }
58
59          ...
60        }
61        this.addEdges(e);
62        this.addVertices(v);
63      }
64    }
65    ...
66  }
```

The first loop passes through all answer sets existing in this `Result` object, the second one traverses all literals in this model. After that it should be checked whether the predicate of this literal represents a highlighted element by searching in the specified `predicatesHigh-lighted` list. If that is the case, then the arity of the literal defines whether it describes an edge or a vertex. Now this element has to be added to the `Edges` or `Vertices` data object along with the highlighted flag.

A similar procedure is done if the literal represents a normal edge and therefore is found in the `predicates` list. The difference is that the edge is added as a non highlighted one and there is no need to add any vertex at all because every answer set is drawn as vertex anyway.

Hint: The `Edges` data object is designed in a way that adding an already existing edge leads to an update, whereas the status flag `highlighted` can only be set from `false` to `true` but not the other way around. That means that an already highlighted edge still remains highlighted.

Every `Edges` and `Vertices` data object that belongs to a single model has to be added as an element of their own to a list in the `VisualizationResult` by calling the `addEdges` and `addVertices` methods. As a result the index of the elements in these lists corresponds with the index of the related answer sets in this `Result` object.

4.8 Presentation as graph

This is the last important step to achieve the desired visualization. We already have all relevant edges, vertices and highlighted elements as data objects that we need to draw a graph. All we have to do is to create a new `Graph` object, add all vertices and edges and finally design the layout. This is implemented in the `GraphVisualization` class. It is a lot of code to define all properties and styles like colors, strokes and shapes. Hence the following example only includes the most important code parts:

```
1  public class GraphVisualization implements ItemListener {
2    private Graph<Integer, Integer> g;
3    private VisualizationViewer<Integer, Integer> vv;
4    ...
5
6    public GraphVisualization(JTextArea textAreaResult, JList jListPredicates)
       {
7
8      ...
9
10     g = new DirectedSparseMultigraph<Integer, Integer>();
11
12     // The Layout<V, E> is parameterized by the vertex and edge types
13     Layout<Integer, Integer> layout = new KKLayout<Integer, Integer>(g);
14
15     layout.setSize(new Dimension(200, 200));
16     vv = new VisualizationViewer<Integer, Integer>(layout);
17
18     vv.setPreferredSize(new Dimension(250, 250));
19     vv.getPickedVertexState().addItemListener(this);
20     vv.getRenderContext().setVertexLabelTransformer(new ToStringLabeller<
         Integer>());
21
22     ...
23
```

```
24     PluggableGraphMouse gm = new PluggableGraphMouse();
25     gm.add(new TranslatingGraphMousePlugin(MouseEvent.BUTTON3_MASK));
26     gm.add(new ScalingGraphMousePlugin(new CrossoverScalingControl(), 0, 1.1f
           , 0.9f));
27     gm.add(new ModPickingGraphMousePlugin<Integer, String>());
28     vv.setGraphMouse(gm);
29 
30     ...
31 
32     Transformer<Integer, Paint> edgePaint = new Transformer<Integer, Paint>()
           {
33       public Paint transform(Integer i) {
34         if(Memory.getInstance().getVisResult().getEdgesOfModel(currentModel).
               get(i).isHighlighted()) {
35           return Color.red;
36         }
37         return Color.black;
38       }
39     };
40 
41     ...
42 
43     Transformer<Integer, Paint> vertexBackground = new Transformer<Integer,
           Paint>() {
44       public Paint transform(Integer i) {
45         if(vv.getPickedVertexState().isPicked(i)) {
46           return Color.yellow;
47         }
48         return Color.white;
49       }
50     };
51 
52     vv.getRenderContext().setEdgeDrawPaintTransformer(edgePaint);
53     vv.getRenderContext().setArrowDrawPaintTransformer(edgePaint);
54     vv.getRenderContext().setArrowFillPaintTransformer(edgePaint);
55     vv.getRenderContext().setEdgeArrowStrokeTransformer(edgeStroke);
56     vv.getRenderContext().setEdgeStrokeTransformer(edgeStroke);
57     vv.getRenderContext().setVertexDrawPaintTransformer(vertexBorder);
58     vv.getRenderContext().setVertexStrokeTransformer(vertexStroke);
59     vv.getRenderContext().setVertexFillPaintTransformer(vertexBackground);
60     vv.getRenderContext().setEdgeShapeTransformer(new EdgeShape.Line<Integer,
           Integer>());
61   }
62   ...
63 }
```

First, an object of `DirectedSparseMultigraph` is instantiated providing a simple directed graph which permits parallel edges. The types of the edges and vertices are defined as integers. Then this graph is added to a new created `KKLayout`, which helps us to organize the vertices in a viewable way. Furthermore, this layout object is added to a new object of `VisualizationViewer`. This viewer represents the main frame in which the graph is embedded.

Now we can set the size, add ItemListener for actions like clicking and set the VertexLabelTransformer to display the integer value of each vertex inside itself. Further on, with the method setGraphMouse an object of PluggableGraphMouse can be set which is responsible for the interactive features like picking, moving and resizing by the user. In order to enable these features it is necessary to add several plugins to the PluggableGraphMouse object: TranslatingGraphMousePlugin is a plugin for moving the complete graph by holding the right mouse, CrossoverScalingControl does the complete zoom feature which can be performed with the mouse wheel. The last plugin ModPickingGraphMousePlugin needed is a modification of the already existing PickingGraphMousePlugin of the *JUNG* library. This is useful for picking one or more vertices and move them around. For this modification, the PickingGraphMousePlugin is adopted from the *JUNG* library and modified in some cases to adjust the interactive mode of the graph like a better multi-selecting feature with the ability to move them around in a more comfortable way.

Now we need some Transformer objects which are added to the VisualizationViewer and are needed for the customized style like colors, shapes and strokes. They are used when a specific element is drawn. For example, when a transformer is set by setEdgeDrawPaintTransformer, then the method transform in this Transformer object will be called each time when the color of an edge is drawn. Of course, in this case the return value has to be a Color object. That way, every edge can be colored individually. In our case, we need a specific implementation to highlight determined vertices and edges by using a different color. Here is the relevant code once again to present the structure of these Transformer objects:

```
1 Transformer<Integer, Paint> edgePaint = new Transformer<Integer, Paint>() {
2   public Paint transform(Integer i) {
3     if(Memory.getInstance().getVisResult().getEdgesOfModel(currentModel).get(
        i).isHighlighted()) {
4       return Color.red;
5     }
6     return Color.black;
7   }
8 };
```

The method Memory.getInstance().getVisResult() on line 3 returns the current VisualizationResult object. This being done the method getEdgesOfModel(currentModel) delivers the Edges data object which contains all edges of the current model. With get(i) we simply get the relevant Edge object because the index i is delivered as argument of the currently drawn edge. The Edge object contains the status which indicates if the edge is highlighted or not (which we have set in the last section when we generated all the edges). We can get this status by the method isHighlighted(). In fact, all highlighted edges are drawn red, all others black.

The same procedure can be applied to the vertices and also can be done with all other layout styles like shapes and strokes.

The only thing left to do is to set the data – all vertices and edges – which is done with an `update()` method:

```
public void update(int index){
  VisualizationResult visResult = Memory.getInstance().getVisResult();
  Result result = Memory.getInstance().getResult();

  Object[] vertices = g.getVertices().toArray();

  // delete all current vertices
  for(int i = 0; i < vertices.length; i++) {
    g.removeVertex((Integer) vertices[i]);
  }

  // add a vertex for each answer set
  for(int i = 0; i < result.getAnswerSets().size(); i++){
    g.addVertex(i+1);
  }

  Edges edges = visResult.getEdgesOfModel(index);
  this.currentModel = index;

  // add all edges which are relevant for this model
  for(Integer id : edges.getAll()) {
    g.addEdge(id, edges.get(id).getFrom(), edges.get(id).getTo(), EdgeType.
        DIRECTED);
  }

  vv.updateUI();
}
```

The argument `index` decides which model should be drawn.

First, all existing vertices have to be deleted and then for every answer set a new vertex with increasing integer value is added to the graph. Finally we receive the current Edges object by calling `visResult.getEdgesOfModel(index)` and then we get all single edges as list by the `edges.getAll()` method. Now we can run through the list and add each and everyone of them to the graph by using `addEdge`.

CHAPTER 5

Conclusion

The *Answer Set Relationship Visualizer* is publicly available for download[1] and can be used to visualize the relationships between answer sets. The simple user interface provided by the wizard allows to execute the visualization process step by step. The configuration window of the software makes it possible to change the current ASP Solver, adjust the number of calculated answer sets and so on. The status bar at the top of the wizard always shows the process steps, also indicating the current status. The execution of the ASP Solver is indicated in a dialog window with the ability to abort this process any time. This can be done because the execution is running in its own thread. The result of every execution is displayed in the next step. Because of this fact the user has always the chance to prove the computed solutions.

In case of the main ASP program the user can filter the necessary predicates to keep the number of new generated facts smaller. In case of the visualization ASP program the edge predicate and highlight edge/vertex predicate can be chosen in a comfortable way. Also multiple selections are possible. The predicates are checked and are only used when they are compatible: unary predicates correspond to vertices, binary predicates to edges.

The final step – the visualization – is kept simple: Left the list of models, in the middle the visualization graph and right the detail box for selected vertices (answer sets). The graph is interactive and provides some useful features like multi-selection, moving the complete graph or only selected vertices and zooming in/out. Therefore, the vertices can be well-arranged. Now the relationships between the answer sets are presented in a comfortable and obvious way. All highlighted vertices and edges which are defined in the corresponding highlight predicate are colored in red, all the others are black. Further on, the result of the main and visualization ASP program can be exported to a simple text file.

[1] http://www.dbai.tuwien.ac.at/proj/argumentation/vispartix/#ARVis

Of course, this software is only a basic module for such a visualization. In future versions some other ASP Wrapper could be implemented or some more properties and configurations could be build into the application. Also, the graph could be extended: for example, appearing text fields at the position of every selected vertex containing the corresponding answer set instead of using the detail box directly to the right. In addition, the export feature could be expanded. Currently, it only exports to text files, which is not very convenient. At first, a better way would be to let the user decide which data should be exported because not all answer sets and informations are necessary at all. Secondly, an addition export format should be PDF. Then it would be possible to export the graph visualization which is the heart of the whole process of course. The result would be much more illustrative and better usable for documentation.

How we see it, there is still a lot to do and potential for future versions. Nevertheless, the current version can be used to handle the main goal – the visualization of the relationships between answer sets – in a very comfortable way.

Bibliography

[1] DLV System. http://www.dlvsystem.com, Accessed: 2012-05-23.

[2] JUNG: Java Universal Network/Graph Framework. http://jung.sourceforge.net/, Accessed: 2012-05-23.

[3] metasp: Complex Optimization in Answer Set Programming. http://www.cs.uni-potsdam.de/wv/metasp, Accessed: 2012-05-23.

[4] Robert Bihlmeyer, Wolfgang Faber, Vincenzino Lio, and Gerald Pfeifer. DLV - User Manual. http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html, Accessed: 2012-05-23.

[5] Piero Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer Set Programming. In Agostino Dovier and Enrico Pontelli, editors, *A 25-Year Perspective on Logic Programming*, volume 6125 of *Lecture Notes in Computer Science*, pages 159–182. Springer, 2010.

[6] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System Description: DLV. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '01, pages 424–428. Springer-Verlag, London, UK, 2001.

[7] Robert Eckstein. Creating Wizard Dialogs with Java Swing. http://java.sun.com/developer/technicalArticles/GUI/swing/wizard/index.html, Accessed: 2012-05-23.

[8] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit Testing in ASPIDE. *CoRR*, abs/1108.5434, 2011.

[9] Camillo Fiorentini, Alberto Momigliano, and Mario Ornaghi. Towards a Type Discipline for Answer Set Programming. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2009.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A User's Guide to gringo, clasp, clingo, and iclingo (version 3.x). http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/guide.pdf/download, Accessed: 2012-05-23.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Challenges in Answer Set Solving. In Marcello Balduccini and Tran Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2011.

[12] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, April 2011.

[13] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs. *CoRR*, abs/1109.4095, 2011.

[14] Nicola Leone and Wolfgang Faber. The DLV Project: A Tour from Theory and Research to Applications and Market. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 53–68. Springer-Verlag, Berlin, Heidelberg, 2008.

[15] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*, AAAI'08, pages 1594–1597. AAAI Press, 2008.

[16] Enrico Pontelli. Answer Set Programming in 2010: A Personal Perspective. In *Proceedings of the 12th international conference on Practical Aspects of Declarative Languages*, PADL'10, pages 1–3. Springer-Verlag, Berlin, Heidelberg, 2010.

[17] Francesco Ricca. A Java Wrapper for DLV. In Marina De Vos and Alessandro Provetti, editors, *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd Intl. ASP 03 Workshop*, volume 78 of *CEUR Workshop Proceedings*. 2003.