Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Visual Design and Analysis Support for Answer Set Programming

VIDEAS

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik: Information & Knowledge Management

eingereicht von

Patrick Zwickl

Matrikelnummer 0525849

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr.techn. Martina Seidl

Wien, 24. Februar 2011

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Visual Design and Analysis Support for Answer Set Programming

VIDEAS

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Informatics: Information & Knowledge Management

by

Patrick Zwickl
Registration Number 0525849

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Dipl.-Ing. Dr.techn. Martina Seidl

Vienna, 24th February 2011 ______________________ ______________________
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Patrick Zwickl, Wiener Straße 12/20, A-2700 Wiener Neustadt

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 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 Quellen als Entlehnung kenntlich gemacht habe.

Wien, 24. Februar 2011

____
Patrick Zwickl

Acknowledgements

Without a fail, it is desirable to define goals for oneself—it is even more desirable to achieve them. On this way to achieve some of my goals, I have been accompanied by many people in a variety of roles, e.g., fellows, supporters, formers, or companions. The challenge of textually capturing my thoughts requires me to focus on a small number standing in an exceeding relationship to this thesis.

Above all, I would like to thank Dipl.-Ing. Dr.techn. Martina Seidl for her enthusiasm for novel ideas, continuos questioning of concepts, and for her consistent collaboration in facilitating a work, whose focus has been intentionally set between classical disciplines. As a matter of course, I would like to address my thanks likewise to O.Univ.Prof. Mag. Dipl.-Ing. Dr.techn. Gerti Kappel for her supervision and advisory work. For the very fruitful and inspiring technical discussions, I would like to thank Ao. Univ.-Prof. Dr.techn. Hans Tompits, Dipl.-Ing. Jörg Pührer, and Bakk.techn. Johannes Ötsch.

Because of the applied support and plenty of patience bestowed, I would like to thank my family and my friends, but in particular Jacqueline Lonsky for her help in proofreading the written matter.

As usual, I would like to pay those an a-priori tribute, who will read a word of this thesis without (moral) duty, but as consequence of their technical or professional thematic enthusiasm. Come what may, I definitely do not glare a glance further than this, and remain with high reliance of the future in the present.

“Ich denke niemals an die Zukunft. Sie kommt früh genug.” (Albert Einstein).

“I never think of the future. It comes soon enough.” (Albert Einstein).

Abstract

In the last decade, logic programming experienced new impetus by the growth of Answer Set Programming (ASP) as one of the key drivers in the academic world. However, ASP could not attract the same interest as other declarative programming languages in practice so far. This lack of interest in ASP may be explained by the absence of a sufficiently supported software engineering methodology resulting in a difficulty of designing and developing ASP programs. No tools supporting the development of ASP programs are available. So far, no modeling environment has been introduced in the context of software development based on the ASP paradigm, which offers valuable abstraction and visualization support during the development process.

This thesis aims at establishing a novel method for visually designing and analyzing ASP programs. Therefore, a graphical approach for the visualization of ASP is proposed, which is based on concepts presented in literature for other declarative approaches. Moreover, concepts of model engineering are combined with the field of logic programming. Following an Model-Driven Engineering approach, an ASP-specific modeling language is established which is able to visualize important facets of ASP. The modeling language is applied within a graphical editor for the model creation. The resulting models are transformed to textual ASP programs by a code generator. The model engineering approaches are used to define the metamodel, a graphical editor, and to generate the ASP program code from models. Therefore, the link between the formalism of ASP and the graphical representation has to be established. Due to the close connection between ASP and deductive databases—databases with logical reasoning capabilities—the widely used Entity Relationship diagram is applied as initial visualization method for ASP programs.

This thesis is structured in iterative advancement phases. Starting with a simple visualization of non-deductive programs, entire solution packages for deductive ASP programs are introduced. In a final step, the technical realization is discussed and evaluated, and potential enhancements are outlined.

Kurzfassung

Im letzten Jahrzehnt erlebte die Logikprogrammierung durch die Verbreitung von Answer Set Programmierung (ASP) als eine der treibende Kräfte neuen Auftrieb in der akademischen Welt. Jedoch konnte ASP in der Praxis bislang nicht das gleiche Interesse hervorrufen wie andere deklarative Programmiersprachen. Dieses fehlende Interesse könnte durch die Absenz geeigneter Methodologien des Software Engineerings erklärt werden, welche die Entwicklung von ASP Programmen erschwert. Tools, die den Entwicklungsprozess von ASP Programmen unterstützen, sind nicht verfügbar. Bislang wurde keine Modellierungsumgebung für die ASP Softwareentwicklung vorgestellt, die eine wertvolle Abstraktions- und Visualisierungsunterstützung bietet.

Diese Arbeit zielt darauf ab, eine neuartige Methode für die Erstellung und Analyse von ASP Programmen zu etablieren. Dies wird durch die Vorstellung eines graphischen Ansatzes für die Visualisierung von ASP erreicht, welche auf Konzepten aus der Literatur für andere deklarative Sprachen basiert. Darüber hinaus werden Konzepte des Model Engineerings mit dem Bereich der Logikprogrammierung verknüpft. Aufbauend auf einem Model-Driven Engineering Ansatz wird eine ASP-spezifische Modellierungssprache erstellt, welche wichtige Facetten von ASP darstellen kann. Diese Modellierungssprache wird durch die Erstellung von Modellen mittels eines graphischen Editors angewandt. Damit erzeugte Modelle können mit einem Code Generator zurück in ASP Programme transformiert werden. Die Model Engineering Konzepte werden eingesetzt um ein Metamodel, einen graphischen Editor und einen Code Generator zu erstellen. Infolgedessen muss eine Verbindung zwischen den ASP Formalismen und der graphischen Repräsentation definiert werden. Aufgrund der engen Verknüpfung zwischen ASP und deduktiven Datenbanken—Datenbanken mit Inferenz-Fähigkeiten—wird das weit verbreitete Entity Relationship Diagramm als initiale Visualisierungsmethode für ASP Programme eingesetzt.

Diese Arbeit verfeinert iterativ eine initiale Lösung. Beginnend bei einer einfachen Visualisierung nicht-deduktiver Programme werden komplette Lösungspakete für deduktive ASP Programme vorgestellt. In einem finalen Schritt wird die technische Realisierung behandelt und evaluiert, und Möglichkeiten für weitere Forschungstätigkeiten in dem Bereich aufgezeigt.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problems . . . 2
1.3 Solution . . . 2
1.4 Structure . . . 3

2 Answer Set Programming **5**
2.1 Syntax . . . 5
2.2 Semantics . . . 8
2.3 Characteristics . . . 9
2.4 Program Examples . . . 11
2.5 Implementations . . . 13
2.6 Summary . . . 16
2.7 Challenges . . . 16

3 Related Work **19**
3.1 Selection Policy . . . 19
3.2 Program Conception . . . 20
3.3 Debugging . . . 26
3.4 Analysis & Verification . . . 28
3.5 Summary . . . 33

4 Visualization of Non-Deductive Answer Set Programs **35**
4.1 The Big Picture . . . 35
4.2 Describing Non-Deductive Answer Set Programs . . . 36
4.3 Code Generation . . . 39
4.4 Evaluation . . . 40
4.5 Critical Discussion . . . 44

5 Visualizing Inferences & Design Decisions **45**
5.1 The Big Picture . . . 45
5.2 Syntactical Preliminaries . . . 47
5.3 Propositional Answer Set Programs . . . 50

5.4 Programs of Literals with Maximum Arity One 58
5.5 Programs of Literals with Unconstrained Arity 69
5.6 Identification of Answer Sets 70
5.7 Summary By Example 75

6 Realization **79**
6.1 Modeling Language 79
6.2 Code Generation 91
6.3 Summary 92

7 Evaluation **93**
7.1 Usage Scenarios By Example 93
7.2 Visualization 101
7.3 Generated Code 105
7.4 Summary 105

8 Lessons Learned: Usability Enhancement **107**
8.1 Optimization Strategies 107
8.2 Visualization of Transitions 108
8.3 Example 110

9 Conclusion **113**
9.1 Summary 113
9.2 Outlook on Advanced Features 114
9.3 Future Work 118

Bibliography **121**

CHAPTER 1

Introduction

1.1 Motivation

Logic programming as application of mathematical logics for programming often is dated back to the proposition of the *advise taker* problem in 1958 [35]. The advise taker problem is a theoretical computer program, which is able to infer new knowledge from a set of given premises. However, the foundation of logic programming cannot definitely be fixed to a certain date. Several decades of active research in this field have passed and until today a lot of facets of logic programming could develop.

In the last decade, logic programming experienced a new impetus by the growth of *answer set programming* (ASP) as key driver [41]. In particular, the success of ASP is caused by allowing programmatic non-monotonic reasoning—the revision of decisions when the knowledge level rises [2]—and a simple definition of general and normative statements [2]. Another asset is its declarative semantics [19] and the existing powerful solvers like DLV and SMODLES (cf. a short introduction of these systems is given in Section 2.5).

The research interest in ASP has been tremendous which is underlined by the number of more than 11 500 publications accessible on ACM Digital Library[1] which relate to ASP. The ASP concept is trimmed towards solving "difficult, primarily NP-hard, search problems" [33]. Moreover, there exists a wide range of research and application areas[2] related to ASP, e.g., decision support systems [3], configuration, information integration, security analysis, agent systems, semantic web, and planning. However, ASP could not attract sufficient interest to be widely integrated in professional application fields and could not even attract the same interest as other logic programming languages such as Prolog[3] [23]. This may be explained by the difficulty of designing and developing ASP programs which is not sufficiently supported by present

[1] Web site: `http://portal.acm.org/`, last accessed: February 24, 2011

[2] More information about these application areas: `http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html`, last accessed: March 3 2010

[3] Access to ISO/IEC 13211-1:1995 (Prolog): `http://www.iso.org/iso/catalogue_detail.htm?csnumber=21413`, last accessed: March 3 2010

tools. The aim of this thesis is to support the transition process from a scientific programming approach—mainly used in theoretical research—to an applicable solution for multitude of different existing problems.

1.2 Problems

The development of ASP programs is hampered by two major issues. First, the consequence of rules might be unclear for the user. This is caused by the multitude of possible rule applications, and the complexity of understanding constraint rules. The non-monotonicity is a great asset, but it aggravates the ease of understanding why a certain answers set is or is not returned. Second, logical sentences are spread over a number of rules. This can be seen as asset for declaring rules as it allows the developer to focus on the most recent issues and to reuse rules. However, it implies a difficulty in overlooking all possible rule matches by implicitly transferring abstract rule definitions to operational rule executions in an ad-hoc manner. Another problem results from the textual definition of ASP programs. The linearity of text files strongly limits the understandability of ASP programs as they themselves represent non-linear graphs.

The stated problems aggravate the ease of designing ASP programs and, therefore, reduces the adoption and quality levels in professional applications. These issues are related with the code-centric approach of the ASP program design process.

1.3 Solution

The problems discussed above are tackled in this thesis by providing a meta-description of ASP programs in such a way that it allows the identification of variable interactions, rule relationships, and their particular application. This is tackled by proposing a graphical approach for designing ASP programs—related to the concept of [16] which presents a formal meta-program of the original program. It is the aim of this approach to set up a layer above classical ASP that is able to establish a design-first paradigm in the development process by hiding code specific elements. Therefore, a model-based design layer unifying the involved artifacts (rules, their literals, and the application of rules) in a single diagram is provided, which allows the designing, analyzing and improving of an ASP program at one single point of interest. This allows the concentration on design- rather than code-specific issues and improves communicability of programs.

This model-based methodology is applied due to following reasons [27]: First, the textual representation of ASP programs linearly and declaratively describes the dependencies of rules and their applicability. Such a description can be compared with a set of relationships which are linearly chained together. Furthermore, the syntax is typical highly influenced by the capabilities of the solver executing the ASP code. By the usage of visualization techniques such linear representations can be graphically organized in a way which provides recognizable relationships identifiable at one single glance. Even the specific adoptions for used solvers can be removed in these graphical representations. In Figure 1.1, an example highlighting the improved understandability of relationships through visualization—a textual rule as unorganized sequential chain of literals on left side, is opposed to an organized visualization on the right side. In the visualization of the literals, *walk*, *swim*, *jump*, and *likes_fish* may be organized around *penguin*

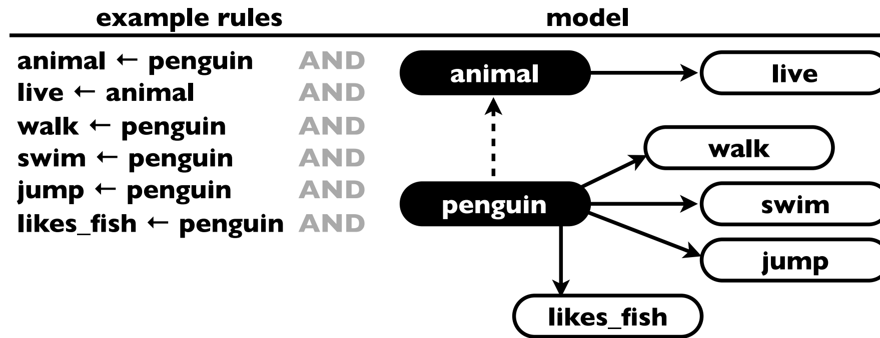


Figure 1.1: Structuring linear text in a non-linear model

which itself inherits *live* from *animal* whereas in the textual representation no similar organization is possible.

This visualization-layer using the described methodology is established in three phases providing the following contributions: (i) A simple structural visualization for non-deductive ASP programs is given. (ii) Starting from propositional logic programs iteratively visualization solutions for deductive n-ary logic programs are constructed. (iii) Subsequently, the modeling language and the associated code generator for the final concept is technically established.

1.4 Structure

This thesis is structured as follows. In Section 2 the fundamentals of ASP are introduced which are complemented by an overview of state-of-the-art ASP solvers. Then related approaches are reviewed in Section 3 and used in Section 4 to provide an initial non-deductive concept for visualizing ASP programs. As major contribution a visualization for deductive programs is given in Section 5. The transition from theoretical concepts to an applicable and executable prototypical solution is described in Section 6 and evaluated in Section 7. From these results some enhancements are proposed in Section 8. In Section 9, this thesis concludes with a summary of results including a brief outlook on the integration of advanced language features.

CHAPTER 2

Answer Set Programming

In this section the syntax and semantics of answer set programs based on the definitions of [14] and [42] are introduced. Moreover, several advanced characteristics, concrete examples, and an overview on state-of-the-art ASP solvers are given in order to introduce the concepts used within this thesis.

2.1 Syntax

Answer set programs are constructed from symbols of their alphabet A which represent the primary artifacts we are concerned with.

Definition 1 (Alphabet) *An alphabet A for defining an answer set program is a triple of the form A=(P, V, C) which consists of the set of* predicates *P, the set of* variables *V, and the set of* constants *C with $P \neq \emptyset$ and $C \neq \emptyset$. The set of constants is also referred to as* domain.

The following naming conventions for elements of an alphabet *A=(P, V, C)* are used within this thesis: Each predicate symbol $p \in P$ is represented as string which has to start with a letter. Each variable $v \in V$ starts with a capital letter, whereas each constant $c \in C$ starts with a lower case letter or a number. Anonymous variables—newly introduced variables which are not reused in a given context—are denoted by an underscore "_".

Some examples of predicates symbols, variables, and constants are as follows:

- *Predicates*: high, bigger, Car, VeryBig, old, fOoD
- *Variables*: X, Y, Month, Building, Year
- *Constants*: x, y, december, empireState, 1920

Definition 2 (Term) *Let A=(P, V, C) be the alphabet of an answer set program where P is a set of predicates, V is a set of variables and C is a set of constants, then an element $t \in V \cup C$ is called a* term. *A term is called ground iff $t \in C$.*

Definition 3 (Arity) *The* arity *n of a predicate symbol* $p \in P$*—denoted by arity(p)—is defined by the number of terms which p is attached to. For each predicate symbol* $p \in P$ *the condition* $arity(p) \geq 0$ *is satisfied.*

A term *t* being attached to a predicate symbol *p* is called an argument of *p*. A sequence of terms $t_1, \ldots, t_n$ is denoted as $[t_1, t_n]$.

Definition 4 (Atom) *Let A be an alphabet of the form A=(P, V, C), then a predicate* $p(t_1, \ldots, t_n)$ *with* $p \in P$ *and arity(p)=n is called* atom *over the alphabet A, iff all* $t_i \in V \cup C$ *(also referred to as atom over p). The atom* $p(t_1, \ldots, t_n)$ *is ground, iff all terms* $[t_1, t_n]$ *are ground. Note that ground atoms may be considered as predicates with arity 0.*

Some examples of predicates with varying arity from *zero* to *k* are as follows. For predicates with arity of zero the brackets may be omitted.

- *Arity 0*: *car()* or *car*, *healthy()* or *healthy*
- *Arity 1*: *car(t), healthy(t)*
- *Arity k*: *airport(*$t_1, \ldots, t_k$*)*

Definition 5 (Literal) *An atom or a negated atom over the alphabet A is called* literal *over A. A negated atom is a symbol of the form* $\neg p$ *(logic negation), where p is an atom. A literal represented by a negated atom is called* negated literal*. Otherwise it is called positive literal.*

According the language features provided by programs, a type differentiation in classic, normal, and extended logic programs is used by [4]. Classic logic programs do not support any negations, whereas normal logic programs extend classic logic programs with *default*-negations (cf. Definition 13). Only extended logic programs (an extension of normal logic programs) support the default-negation (*not*) as well as the strong negation ($\neg$). Extended logic programs are considered in this thesis. However, *default*-negations represent monadic connectives [4] and are, therefore, not contained in literal definitions.

Answer set programs consist of rules which hold a set of literals. According the uses head literals a differentiation in disjunctive and non-disjunctive rules is made (cf. Definition 6). Moreover, rules can represent facts or constraints, or can be positive, false, or ground.

Definition 6 (Rule) *A* rule *r over the alphabet A is a tuple* $\langle H, B \rangle$ *where H is a set of atoms and B is a set of literals over A. The rule r is* disjunctive *iff* $|H| > 1$*. Iff* $|H| = 1$*, the rule is called* non-disjunctive*. The disjucntive rule r with head {*$h_1, \ldots h_k$*} and body {*$b_1, \ldots, b_n$*, not* b_{n+1}*, ..., not* b_m*} is denoted by*

$$h_1 \vee \ldots \vee h_k \leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m.$$

One exemplary disjunctive and one non-disjunctive rule is given in the following:

- *Disjunctive*: $warm \vee hot \leftarrow summer, sun.$

- *Non-disjunctive*: $warm \leftarrow summer, sun.$

Definition 7 (Rule types) *A rule r over the alphabet A is*

- *a* fact, *iff* $body(r) = \emptyset$ *where* $body(r)$ *are the all body literals of r;*
- *an* (integrity) constraint *(here named constraint rule), iff* $head(r) = \emptyset$ *where* $head(r)$ *are all head literals of r;*
- positive, *iff* $body^{-}(r) = \emptyset$ *where* $body^{-}(r)$ *represents all negated literals of the body (body literals);*
- ground, *iff every atom of head(r) and body(r) is ground;*
- false, *iff* $body(r) = \emptyset$ *and* $head(r)$=$\emptyset$ *;*

If r is neither a constraint nor a fact rule, it is named standard rule.

Definition 8 (Program) *An answer set program* Π *is a set of rules according to Definition 7.*

The *Herbrand Base* (*HB*) for the answer set program Π is implicitly defined by the alphabet *A* of Π. The *Herbrand Universe*—$HU(\Pi)$ for the answer set program Π—is the set of ground terms formable with the alphabet *A* of Π [2]. The Herbrand Base—$HB(\Pi)$—is the set of ground atoms formable with the predicates *P* of *A* of a program Π [2] and the arguments of $HU(\Pi)$.

$$\Pi_{2.1} = \begin{cases} r1: parent(terach, isaac) & \leftarrow . \\ r2: parent(isaac, abraham) & \leftarrow . \\ r3: ancestor(X,Y) & \leftarrow parent(X,Y). \\ r4: ancestor(X,Z) & \leftarrow parent(X,Y), ancestor(Y,Z). \end{cases} \tag{2.1}$$

The program $\Pi_{2.1}$ based on [47] describes the *ancestors* of people from known parent-child relationships (*parent*). From the definition of $\Pi_{2.1}$, its Herbrand Universe *HU(*$\Pi_{2.1}$*) = {terach, isaac, abraham}* and the predicates $P_{2.1}$*={parent/2, ancestor/2}*—where *parent/2* and *ancestor/2* represent predicates with the arity two—can be identified. With the use of $P_{2.1}$ and *HU(*$\Pi_{2.1}$*)* the Herbrand Base HB is given by*{*

parent(terach, terach), parent(terach, isaac), parent(terach, abraham), parent(isaac, isaac), parent(isaac, terach),parent(isaac, abraham), parent(abraham, abraham), parent(abraham,terach), parent(abraham, isaac),

ancestor(terach, terach), ancestor(terach, isaac), ancestor(terach, abraham), ancestor(isaac, isaac), ancestor(isaac, terach), ancestor(isaac, abraham), ancestor(abraham, abraham), ancestor(abraham, terach), ancestor(abraham, isaac)

} is constructed.

2.2 Semantics

The answer set semantics as extension of the stable model semantics [18] of normal logic programs defines the semantics for answer set programs.

Definition 9 (Interpretation) *A set of positive and strong negated literals is an interpretation X, iff X does not contain any complementary literal pairs. X* satisfies *a literal L_i (denoted by $X \models L_i$), if $L_i \in X$ and $\neg L_i \notin X$. A rule r is* applicable *under X, iff $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$—otherwise it is* blocked *under X. The rule r is satisfied by X ($X \models r$), iff $L_i \in head(r)$ and $L_i \in X$ or r is blocked under X. X satisfies a program Π over A($I \models \Pi$), iff X satisfies r for every rule $r \in \Pi$.*

Definition 10 (Model) *The interpretation X is a* model *of Π, iff $X \models r$ for each rule $r \in \Pi$. The model X of Π is minimal for Π, iff there does not exist any model X' of Π with $I' \subset I$.*

$$\Pi_{2.2} = \begin{cases} r1 : car & \leftarrow . \\ r2 : red & \leftarrow \text{car.} \\ r3 : traffic & \leftarrow \text{car.} \end{cases} \tag{2.2}$$

$$M_{2.3} = \begin{cases} m0 : \{car, red\} \\ m1 : \{car, red, red, traffic\} \\ m2 : \{car, red, traffic\} \end{cases} \tag{2.3}$$

In Equation 2.3 three interpretations are given for $\Pi_{2.2}$. The interpretations $m1$ and $m2$ are models of $\Pi_{2.2}$ as they satisfy each rule $r \in \Pi_{2.2}$—whereas $m0$ is not a model of $\Pi_{2.2}$. As $m2 \subseteq m1$ holds, $m1$ cannot be a minimal model. $m2$ is a minimal model of $\Pi_{2.2}$, because no model X for $\Pi_{2.2}$ exists for which $X \subseteq m2$ and $X = m2$ holds.

Definition 11 (Answer set) *Each minimal model X of Π is an answer set of Π denoted by AS(Π). Iff AS(Π) = $\emptyset$ ($|AS(\Pi)| = 0$), then Π is called inconsistent. Each $A \in AS(\Pi)$ is a subset of the Herbrand Base of Π.*

The answer sets of an ASP program Π are identified by computing the reduct Π^S of the ground instantiated program Π for the state S (S is an interpretation consisting of a set of literals $L \in \Pi$). This computation is based on the Gelfond-Lifschitz-Reduction [18] as described in Definition 12 [4]. In particular, each model *X* is an *answer set* of Π, iff X is a minimal model of the *reduct* Π^X of Π with $\Pi^X = \{\ head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\ \}$.

Definition 12 (Reduct) *A* reduct Π^S *for the state S of the program Π is of the form: $\Pi^S := \{H \leftarrow A_1, \ldots, A_n. \mid H \leftarrow A_1, \ldots, A_n, notB_1, \ldots, notB_m. \in \Pi, \{B_1, \ldots, B_m\} \cap S = \emptyset\}$. Π^S is computed for Π by eliminating all rules, which contain a default-negated body literal not B where $B \in S$, and by eliminating all negated body literals from all other rules.*

The default and logical-negation allow the differentiation between queries failing as a consequence of undemonstrability (not)—the satisfaction of the query is unprovable—and queries failing by proving the negation of the query [4].

Definition 13 (Negation) *If for a model X of* Π *and the literal L the conditions* $X \not\models L$ *and* $X \not\models \neg L$ *hold, then undefined(L) for X and* $X \models$ *not L hold (*default-negation *of L). If for a model X of* Π *the condition* $X \models \neg Q$ *holds (*strong negation *of Q), then* $X \not\models Q$ *and* $X \models$ *not Q hold as well.*

Constraint rules eliminate integrity breaking models from the set of answer sets $AS(\Pi)$ for an answer set program Π. Consequently, no answer set $X \in AS(\Pi)$ can violate any constraint rule *r* of the form *body(r)={*$b_1, \ldots, b_k$*}* and *head(r)=*$\emptyset$. Fact rules add tuples which hold for all answer sets of $AS(\Pi)$. Therefore, each answer set *X* of the program Π has to satisfy $X \cap head(r) \neq \emptyset$ for every fact rule $r \in \Pi$ of the form *head(r) = {*$h_1, \ldots, h_k$*}* and *body(r)=* $\emptyset$.

The answer set program $\Pi_{2.4}$ provides two answer sets: $\{tree, big\} \models \Pi_{2.4}$ and *{bush,* $\neg$ *big}* $\models \Pi_{2.4}$. Rule *r1* is a fact which determines that one of disjunctively connected literals is part of each answer set—*tree* $\vee$ $bush$. The rule *r3* uses the default-negated body literal *not big* which is satisfied whenever an answer set does not contain *big*. As a consequence, whenever a answer set contains *bush* instead of *tree* as consequence of applying rule *r1*, rule *r2* cannot be applied and *big* does not hold. Therefore, the body literals of rule *r3* can be satisfied by this answer set, which leads to the satisfaction of $\neg big$—the *bush* is, therefore, definitely not big. For trees it is obviously the opposite situation.

$$\Pi_{2.4} = \begin{cases} r1 : tree \vee bush & \leftarrow . \\ r2 : big & \leftarrow tree. \\ r3 : \neg big & \leftarrow \text{not } big. \end{cases} \tag{2.4}$$

2.3 Characteristics

The following properties for answer set programs are discussed in this section: First, characteristics of executing ASP programs are reviewed. Second, two language characteristics are inspected—the non-monotonicity and the declarativity of ASP.

Execution of Answer Set Programs

According to [2] there exist several characteristics concerning the execution of ASP programs, which are listed in the following:

- In answer set programs the ordering of literals within the head or the body of a rule is not important for the outcome. This stands in high contrast to other logic programs where a certain literal application order is defined which influences the computed results.

- The same applies for the ordering of rules. In answer set programs, no ordering of rules is defined. An ordering of the rule application $r_1, \dots, r_i, \dots, r_j, \dots, r_n$, therefore is not guaranteed. If both rules r_i and r_j are eligible to be applied, there is no guarantee that r_i is applied before r_j or vice versa. However, practically the answer set solvers have to apply one rule after each other.
- The *default*-negation allows a consistent regulation of loops and recursions.

Non-Monotonicity

An essential property of answer set programming is its *non-monotonic* nature. This means that decisions are revised in ASP programs Π when the knowledge is isolated. For example, the knowledge of *A* can be used to infer *B*. However, if we do not only know *A*, but *C* as well, it could eliminate or modify the previously computed result.

Definition 14 (Non-monotonocity) *A program* Π *is monotonic, iff for every* X_i *for which* $\Pi \models X_i$ *holds,* $\Pi \cup r \models X_i$ *holds as well for any arbitrary rule r—otherwise* Π *is* non-monotonic. *A language is non-monotonic, iff it allows the definition of any non-monotonic program* Π*—otherwise the language is* monotonic.

$$\Pi_{2.5} = \begin{cases} r1 : summer \lor winter & \leftarrow . \\ r2 : warm & \leftarrow oven_on. \\ r3 : warm & \leftarrow summer. \\ r4 : \neg icy & \leftarrow warm. \\ r5 : & \leftarrow \neg icy, winter, oven_on. \\ (r6 : oven_on & \leftarrow .) \end{cases} \tag{2.5}$$

For example in Equation 2.5 rule *r1* is a fact of the form *summer* ∨ *winter*. In this example it is *warm* in *summer* or when the oven is on (rule *r2* and *r3*). Whenever it is *warm* it cannot be *icy* (rule *r4*). Rule *r5* asserts that it is not possible to believe that $\neg icy$ has to hold when *warm* is inferred of consequence of *oven_on*. This represents a revision of previous decisions. If the oven is turned on, e.g., by uncommenting rule *r6*, this revision is necessary to avoid inconsistent states, e.g., a system of a car believes it cannot be icy, although it is icy. The most common way of receiving inconsistent states is the massive usage of default-negations and disjunctions.

Declarativity

Answer set programming is a fully declarative logic programming language [2]. Declarative languages provide declarations built of symbols that implicitly define a certain associated operational behaviour that is applied by underlying runtime solver. Declarative programs are, therefore, a set of declarations D_i representing the behaviour of Π and the associated objects to be reasoned about—$\Pi = \bigcup_{i=0}^{n} D_i$ for the program Π. Each declaration D_i requires a runtime

interpretation which dynamically associates imperative commands in the form $C_i \rightarrow C_{i+1} \rightarrow \ldots \rightarrow C_L \rightarrow \ldots \rightarrow C_n$ with D_i. These commands can be different for each solver S interpreting D_i. For this purpose, D_i needs not express the operational commands associated with Π and, therefore, allows an easier specification of Π. However, the impedance mismatch of designtime and runtime have to be independently faced for each used tuple $\langle \Pi, S \rangle$ (where S is an arbitrary solver), as every S can use different algorithms interpreting the declarations of Π.

Definition 15 (Declarative behaviour) *B is the behaviour associated with Π which results from $B_\Pi = B_D + B_S$ where B_D is the behaviour specified by the declarations $D \in \Pi$ and B_S is the behaviour resulting from the program interpretation of a solver S.*

An expected behaviour B and a set of objects O have to be formulated in declarative statements to allow reasoning. These declarations, furthermore, remove the necessity of specifying all details of B. For example the literal *buy(x,y)* with arity two could mean that *x* buys product *y*. It cannot specify the details of involved actions (commands), e.g., *x* goes to shop *s* by using bus *b*, looks for product type *p* in row *r*, finds it in cell *c*, and buys it with his credit card *u*. In lieu of an expression providing the operational semantics, only declarations are placed hiding the real application behaviour. This allows the focussing on the design of the rules rather than on the application of the rules and the implied actions.

Another characteristic is the differentiation of hard failing (negation) and failing caused by missing evidence (result is unknown). This behaviour has already been described in previous sections.

2.4 Program Examples

In this section some examples of answer set programs are presented to put the previously given definitions into practice. Beginning with a simple program, incrementally further ASP features are reviewed. In particular, the negations, the splitting in cases, the revision of decisions, and the usage of functions are discussed.

Negations

The program $\Pi_{2.6}$—adopted from [4]—uses positive and negative literals. The constant *polly* refers to a bird whereas *tweety* refers to a penguin—as we know penguins are birds that cannot fly. The program $\Pi_{2.6}$ describes this problem as follows. Rule *r1* asserts that each penguin is a *bird*. Rule *r2* defines that every bird can fly, if not defined differently. Without rule *r3* *tweety* could fly as well, although he is defined as penguin by *penguin(tweety)*. Rule *r3* fixes this problem by specifying that penguins cannot fly. Rule *r2* and *r3* may look contradictory at the first glance, but this combination of rules specifies that flying penguins cannot be contained in a stable answer set fulfilling all rules. The only resulting answer set, therefore, is *{penguin(tweety), bird(polly), bird(tweety), ¬ fly(tweety), fly(polly)}*.

$$\Pi_{2.6} = \begin{cases} r1: bird(X) & \leftarrow penguin(x). \\ r2: fly(X) & \leftarrow bird(x),\ \text{not}\ \neg fly(x). \\ r3: \neg fly(x) & \leftarrow penguin(x). \\ r4: penguin(tweety) & \leftarrow . \\ r5: bird(polly) & \leftarrow . \end{cases} \tag{2.6}$$

Cases & Revisions

For many ASP programs several different answer sets exist—these answer sets represent different valid solutions for applying the rule set of a program. Program $\Pi_{2.7}$ based on the idea of [4] highlights that the default-negation is a key driver for increasing the number of such cases—the number of returned answer sets.

$$\Pi_{2.7} = \begin{cases} r1: \neg R(x) & \leftarrow P(x),\ \text{not}\ R(x). \\ r2: P(X) & \leftarrow \text{not}\ Q(X). \\ r3: Q(X) & \leftarrow \text{not}\ P(X). \end{cases} \tag{2.7}$$

In particular, the rules *r2* and *r3* are complementary as the one is applied only, if the result of the other rule has no evidence. Both *P(X)* and *Q(X)* cannot be part of the same answer set. Applying rule *r2* before *r3* returns *P(X)*, as there is no evidence of *Q(X)*. As a consequence the body of *r3* is not fulfillable any more. The opposite occurs, if the rule *r3* is applied before. As the order of execution is not defined in the answer set semantics, both answer sets are returned—one holds *P(X)* and one *Q(X)* with all consequences resulting from the eligibility of applying other rules.

Another important aspect is the revision of decisions for which constraint rules are key drivers. The usage of such rules can be seen in Equation 2.5. The primary difference to standard rules is the semantics of constraint rules. Constraint rules (cf. rule *r5* in Equation 2.5) are used to eliminate impossible predicate combinations in answer sets, but not to infer new knowledge. Sometimes classical rules can have a similar functionality—e.g., the usage of contradictory rules *r3* and *r4* in Equation 2.7 disallows that both predicates are used in the same answer sets.

Contradictions

Programs can be constructed, which make us of contradictory rules (complementary rules) leading to no stable answer set. In the program $\Pi_{2.8}$ [17] the rule *r1* adds $\neg p(X)$ to each answer set, which does not contain *p(X)*. The rule *r2*, however, adds $p(X)$ to the answer set, whenever $\neg p(X)$ is contained. This represents a contradiction, as the literal $p(X)$ and its negation are contained in the same candidate answer set. Consequently, no valid stable answer set can result from $\Pi_{2.8}$.

$$\Pi_{2.8} = \begin{cases} r1 : \neg p(X) & \leftarrow \text{not} p(X) \\ r2 : p(X) & \leftarrow \neg p(X). \end{cases} \tag{2.8}$$

Functions

Not only static values are applicable as literal arguments, but also functions and functional operators can be added in certain implementations. The program $\Pi_{2.9}$ [17] for example shows the possibility of functional extensions in ASP.

$$\Pi_{2.9} = \begin{cases} r1 : e(0). & \\ r2 : e(X+2) & \leftarrow \text{ not } e(X). \\ r3 : p(X+1) & \leftarrow e(X), \text{ not } p(X). \\ r4 : p(X) & \leftarrow e(X), \text{ not } p(X+1). \end{cases} \tag{2.9}$$

If no maximum number is set in the solver, an infinite number of answer sets containing an infinite number of literals will result from $\Pi_{2.9}$, e.g., *{e(0), e(3), e(4), e(7), e(8), ...}* [17]. Concrete implementations of solvers are reviewed in the following section.

2.5 Implementations

To be able to compute the answer sets from a program Π so called *solvers* (solver systems) are used. Solvers are implementations for answer set programs following the answer set semantics presented above. In this section, state-of-the-art ASP solvers are introduced and compared. According to [4] all introduced systems act by searching all possible states. As this proceeding involves a high effort, efficient algorithms are necessary which optimize the following tasks [4]: (i) the actual searching algorithm, (ii) the reduction of the search space, and (iii) the algorithm for grounding.

Non-Disjunctive Solvers

There exists a huge set of answer set solvers that do not support disjunctions. In the following only two very popular and wide-spread solvers are introduced which are useable by the front-end Lparse[1]—namely SMODELS and ASSAT(X). Such front-ends are necessary to allow the developer to access the functionality provided by a solver library through a graphical user interface or command-line utility.

[1]SMODELS & Lparse information and download: `http://www.tcs.hut.fi/Software/smodels/`, last accessed: February 24, 2011

SMODELS

SMODELS[1] is the most popular system for answer set programs [31]. It is a non-disjunctive solver following the answer set semantics extended with several additional features like functions, cardinalities, and weights [31].

An example of cardinalities provided by [31] is as follows:

$$1\{a, b,\ \text{not}\ c\}2.$$

These cardinalities can be seen as boundaries for literals—i.e., a minimum and maximum number of body literals is given, which have to be contained in a candidate answer set in order to apply a rule. In the example, one is the lower and two is the upper boundary. This indicates a minimum of one and a maximum of two literals, which are allowed to be satisfied. If in the example above a and b are satisfied, c may be satisfied despite the default-negation as well (in the knowledge of Π) in order to be able to apply this rule. If c is not known to be satisfied, *not* c is fulfilled. This would exceed the upper boundary.

Cardinalities implicitly give each addressed literal the same weight—only the number of literals contained in the answer set is relevant. In contrast the concept of weights extends cardinalities by explicitly assigning weights to each literal. A weight is some numerical value, typically a decimal number in the interval [0,1]. The lower and upper bound are expressed by decimal numbers as well which state which sum of literal weights at least and at most has to be satisfied.

An example of weights is provided by [31]:

$$1.02 \leq \{a = 1.0, b = 0.02,\ \text{not}\ c = 0.04\} \leq 1.03.$$

ASSAT(X)

In this section the Answer Set by SAT[2] (ASSAT(X)) solver is introduced (cf. [31,34]). It is based on the use of satisfiability (SAT) solvers. SAT solvers are widely adopted and highly optimized systems. ASSAT(X) can be used with different SAT solvers. The overall process involves the transformation of a program Π to a set of clauses which is then computed by the SAT solvers to produce a single output model M. If the solver is able to verify that M is an answer set, this is the result. If it is no answer set, the set of clauses is extended [31]. As it only searches for a single output model, it can only compute one answer set at once. For computing other sets, rules have to be added that can exclude already retrieved answer sets [34]. This, obviously, restricts the ease of use.

Disjunctive Solvers

There exists a set of solvers that is able to handle disjunctions in answer set programs. Two of these solvers are introduced in this section—namely DLV and CMODELS. Disjunctions are valuable in ASP as they allow the specification of more general rules that allow the consideration of different cases (answer sets) within a single rule.

[2] ASSAT(X) download and information: `http://assat.cs.ust.hk/`, last accessed: February 24, 2011

DLV

The DLV project[3] realizes a Disjunctive Datalog System [31]. DLV is an extension of the Datalog language to allow the usage for answer set programming—it integrates the epistemic *or* connective. DLV is free for academic and non-commercial use. It is a generic system consisting of a kernel on which several front-ends are built on. The built-in front-ends provide special functionalities for inheritance, diagnosis, planning, database querying, and meta-interpretation [31]. A set of external front-ends is available as well.

DLV allows the usage of disjunctions [32]. DLV furthermore allows the usage of queries and is intended to be used for the Guess/Check/Optimize (GCO) paradigm [31]—the GCO holds rules for guessing an output, constraints for checking integrity of solutions, and an optimization part using "weak rules" [31].

The DLV kernel uses as three-layered architecture to compute results. The main steps of the computation involve the usage of "heuristics with extensive lookahead" [31] to compute models. These models are then verified by model-checkers to show, if they are answer sets of this program or not. Found answer sets are then returned.

$$\Pi_{DLV} = \begin{cases} r1 : -ok & : -not - hazard. \\ r2 : male(X) \vee female(X) & : -person(X). \\ r3 : fruit(P) \vee vegetable(P) & : -\mathit{plant_food(P)}. \\ r4 : true \vee false & : -. \end{cases} \tag{2.10}$$

An exemplary program provided by [5] using the DLV notation is shown in Equation 2.10. It demonstrates the abilities in specifying disjunctive connectives. For example fact rule $r4$ infers either $true$ or $false$. As no other rules are used that could avoid this result, at least two answer sets have to be returned just from the specification of this program. Other disjunctions are placed in $r2$ and $r3$. The separation of head and body ($\leftarrow$) is shown as $: -$ which can be easier typed. The logical-negation $\neg$ is denoted by a hyphen prior to a literal, e.g. *–hazard*.

CMODELS

Similar to ASSAT, the CMODELS[4] solver makes use of SAT solvers as well. However, with the support of the solver zChaff[5]—which is a SAT solver as well—it can proof the "minimality of found models"[4]. The SAT solvers are used for searching models M that are answer sets. CMODELS makes use of the Lparse front-end as well. It is intended to be used for the computation of answer sets for programs "that are tight or can be transformed into tight programs, and does not suffer from these limitations" [32]. Tight programs do not hold any loop formulas [32]

[3]DLV download and information: `http://www.dbai.tuwien.ac.at/proj/dlv/`, last accessed: February 24, 2011

[4]CMODELS download and information: `http://www.cs.utexas.edu/~tag/cmodels/`, last accessed: February 24, 2011

[5]zChaff information: `http://www.princeton.edu/~chaff/zchaff.html`, last accessed: February 24, 2011

which could avoid termination [34]. In the second release of CMODELS called CMODELS-2, the computation of non-tight programs works similar to ASSAT [32]. It, therefore, "is capable of handling arbitrary nondisjunctive programs, by implementing the same techniques as ASSAT" [31].

Summary of Comparison

There exists a set of ASP solvers which support different language constructs like disjunctions, and computation approaches, but are similar in syntax. Often different solvers can be integrated in a front-end, e.g., Lparse, and even the base solvers can often be extended. There are even solvers that can only compute one single answer set for Π—e.g., ASSAT(X). Not all solvers are able to handle disjunctions (epistemic or). Many examples in previous sections have used disjunctions and, therefore, it is for the context of this thesis highly desirable to use systems with such a functionality. Interesting extensions are available in the SMODELS solvers, e.g., cardinalities.

2.6 Summary

Answer set programs are constructed by an implicitly defined alphabet $A = (P, V, C)$ where P is a set of predicates, V a set of variables, and C a set of constants. From A a set of rules are constructed which are can be *disjunctive* or *non-disjunctive*. Each answer set program Π follows the answer set semantics which is an extension of the stable model semantics. The reduct Π^X of Π, the Gelfond-Lifschitz-Reduction [18], of every minimal *X* is an answer set of Π denoted by $X \subseteq AS(\Pi)$. Answer set programs as extended logic programs support the *default*-negation *not*, failing for undemonstrability, and the logical-negation ¬. Moreover, answer set programs Π are of declarative nature, i.e., they hide the operational application of Π, and non-monotonic nature, i.e., they increase the knowledge state can revise decisions. The operational application of Π is achieved by using answer set programming implementations which are called solvers, e.g., DLV (disjunctive) or SMODELS (non-disjunctive).

2.7 Challenges

In this section, some challenges are discussed, which have to be solved for making ASP attractive in practical application. The first challenge is illustrated in Equation 2.11—addressing the 3-Coloring-Problem. In particular, the program $\Pi_{2.11}$ colors each vertex (*vtx*) in *red*, *green*, or *blue*. Vertexes connected by an *edge* should not have the same colour which is avoided by the specification of constraint rule *r6*. However, the result might not meet the expectations of most developers as one error exists. In rule *r3* the variable *U* is unbound—unbound means that a variable is used once in a single context. This is an error which has to be corrected by replacing it with *V*. This example is given to highlight the great effect a simple statement or hardly recognizable typing mistake can have for the behavior of Π.

$$
\Pi_{2.11} = \begin{cases}
r1a : edge(b, a) & \leftarrow . \\
r1b : edge(c, a) & \leftarrow . \\
r2a : vtx(a) & \leftarrow . \\
r2b : vtx(b) & \leftarrow . \\
r2c : vtx(c) & \leftarrow . \\
r3 : chlrd(V, red) \vee chlrd(V, green) \vee & \\
\vee chlrd(V, blue) & \leftarrow \text{vtx(U).} \\
r6 : & \leftarrow \text{edge(V, U), chlrd(V, C),} \\
 & \quad \text{chlrd(U, C).}
\end{cases} \tag{2.11}
$$

This appetizer is considered to provide the first challenge to be solved by this thesis, i.e., the easier identification of erroneous code. The second challenge is the difficulty of understanding dependencies resulting from deductions in ASP programs. This challenge is also based on program $\Pi_{2.11}$ of Equation 2.11 in which the recognition of relationships resulting from deductions or structural conditions such as from *vtx* and *edge* to *chlrd* are important for the overall program behavior. These relationships also include type and cardinality constraints which are textually only implicitly introduced and are, therefore, difficult to be handled.

CHAPTER 3

Related Work

To overcome the already discussed problems of answer set program development (cf. Section 1), a development support methodology is necessary. The development process is divided into three major tasks (extended ideas from [49]): (i) The initial development of a solution for a given problem and the iterative improvement of the program quality and functionality, (ii) the debugging process (finding the reasons for errors and removing them), and (iii) a verification of the formulated solution against its specification. The focus of these three tasks within this thesis is mainly set on (i), as the finding of errors or even the verification of solutions has to take place after formulating an initial solution for a problem. Optimally, a single approach is capable of supporting all three tasks, but not a single integrated solution for ASP was available at the moment of writing this thesis. For this reason, this section aims at identifying concepts with the closest available relationship to ASP supporting at least one of the three tasks.

3.1 Selection Policy

The focus of the chosen approaches is set on visualization techniques, in order to profit from the capability of visualizations to meaningfully structure solution dependencies in a parallel manner. Such a structuring reduces the effort in recognizing the underlying program behavior and, therefore, supports the developer in the software development process. Although many visualization approaches exist, most of them are not relevant for ASP. Additionally, the related work focus has to be generalized and extended to all languages of declarative nature, as approaches for assisting the ASP development are very rare.

As a consequence, mainly visualizations assisting the software development of declarative languages are introduced which are supplemented by a few relevant visualization techniques for imperative languages. However, most imperative visualization concepts are not transferrable to the field of declarative programming and are, therefore, not relevant. In particular, the Unified Modeling Language (UML) is fully trimmed to the object-oriented paradigm which is not related to ASP. Though, a solution is presented in this section referring to concepts of MOF and UML, respectively.

3.2 Program Conception

One aspect for supporting the software development is the assistance in the design and conception process of programs. The program conception refers to the process of identifying solution paths for a given problem and for finding concepts for meaningfully structuring and modularizing these programs. In particular, two main categories of such solution are of interest. First, concepts from the data and knowledge engineering field, e.g., database modeling techniques, are given to inspect classical solutions which have already been established or have the potential to be established as standard for the program conception process. Thereafter, this section continues with a set of logic-oriented approaches being closer related to ASP.

Data & Knowledge Engineering

In the field of data and knowledge engineering well known and established concepts exist for visually supporting the conception of programs and databases. Especially, the *Entity Relationship* (ER) [10] diagram dominates the definition of data structures today which is highly interesting as ASP predicates can be compared to data structures as well. Furthermore, the visualization of program elements such as rules has a high importance in understanding and designing an ASP program. Consequently, a solution for modeling ontologies is given which is capable of visualizing rules—these rules are similar to ASP rules.

Entity Relationship Diagram

Answer set programs have a close relationship to deductive databases [15]—databases with logical reasoning capabilities. The design process of databases mainly focuses on defining the database schemas. These schemas are mostly modeled by using ER-diagrams which allow a clear definition of existing structures and dependencies. For example in Figure 3.1 the two tables *Person* and *Address* are designed with their *attributes* as *columns*, e.g., svnr or ID, and their associated data types, e.g., String, by the means of a standard ER-diagram. Each table in the ER-diagram may be created by SQL statements. These statements are used for physically establishing or describing the database structures.

There even exist relevant extensions to ER-diagrams such as ERL [22] or the approach presented in [21] which add deduction functionalities to ER-diagrams: ERL is a query language allowing the formulation of clauses in predicate logic visualized as ER-diagrams [13]. From these clauses—i.e., assertions—an ERL-graph can be constructed which visually represents the deductions expressed by these clauses.

The Figure 3.2 provided by [22] demonstrates the capability of ER-diagrams to model structural relationships between entities, e.g, the binary relationship between *Science Fac Student* and *Science Faculty Enrollment*. However, inferences and constraints—such as ensuring that *Suspended Student* and *Science Fac Student* are disjoint—cannot be modeled in classical ER-diagrams and, therefore require assistance by ERL and the ERL-graph respectively. In Figure 3.3 this example is transferred to an ERL graph where a set of clauses for an ER-diagram—representing a faculty organization—are visualized. The ERL graph complements the function-

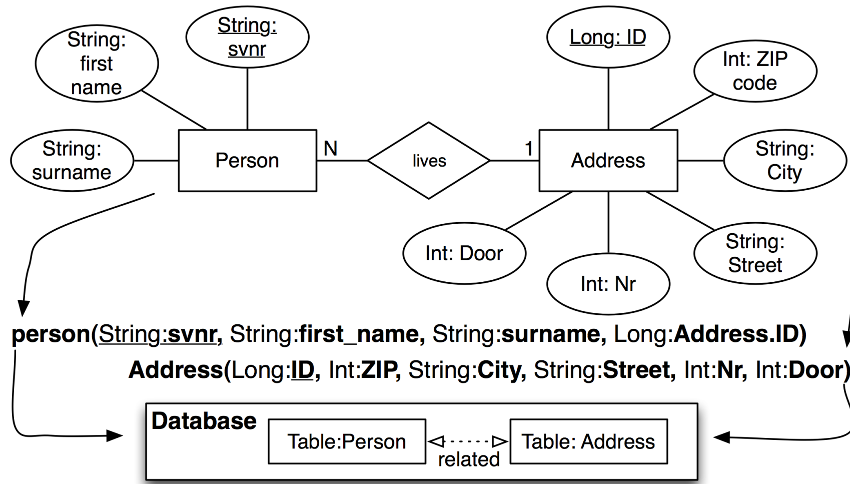


Figure 3.1: Data structures relating to database structures shown in an ER-diagram

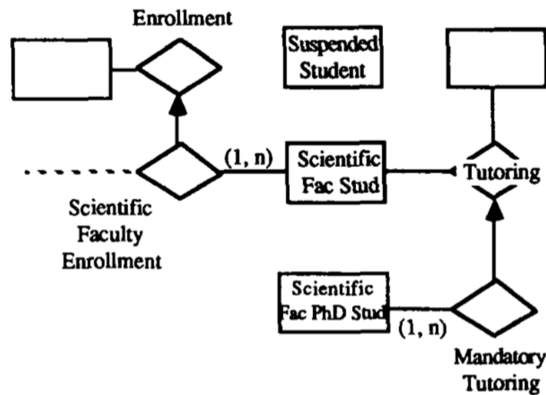


Figure 3.2: A sketched ER-diagram being used as basis for an ERL graph

ality provided by ER-diagrams by providing inference capabilities through the visualization of clauses.

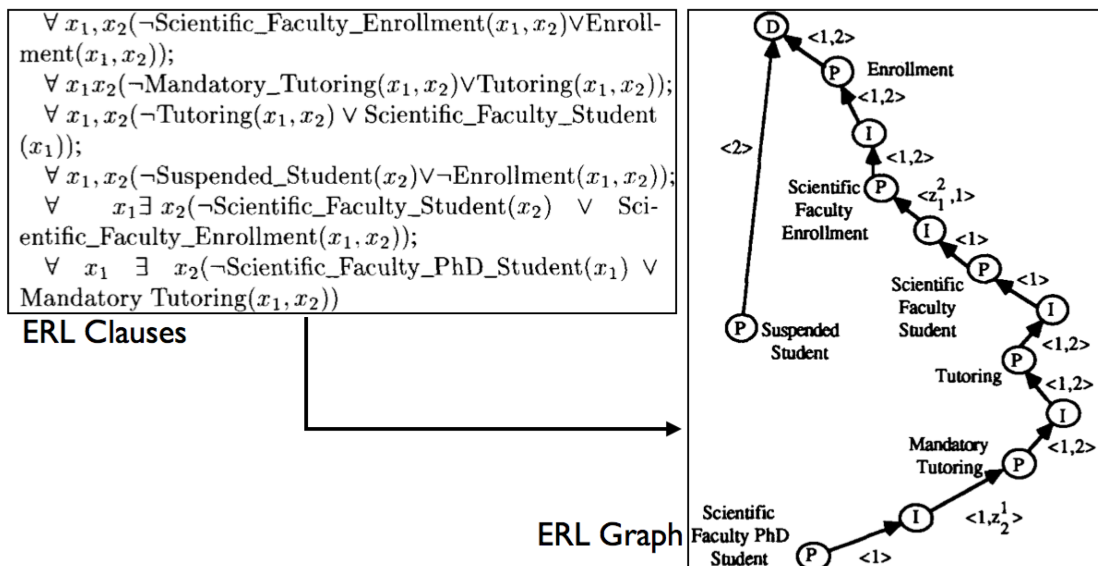


Figure 3.3: Transforming ERL clauses for an ER-diagram to an ERL-graph

Another approach is given by [21] proposing an extension for ER-diagrams allowing the formulation of constraints within the diagram to avoid inconsistent states (these constraints are comparable to ASP constraints).

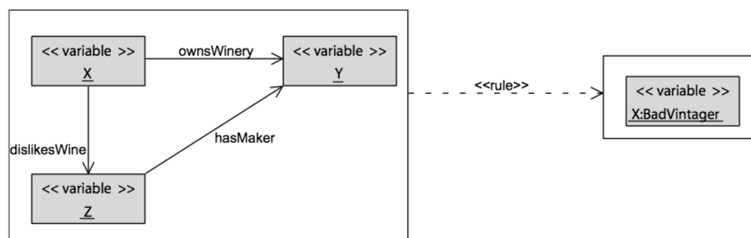
The above given ER-diagrams highlight the good capabilities of ER-diagrams for graphically organizing relationships. For this reason, ER-diagrams established themselves as fitting technique for the initial conception and iterative enhancement of databases. However, it has not proved suitable for visualizing entire logic programs yet as only query language and constraint extensions already exist. Nevertheless, an advancement of ER-diagrams to model ASP programs according the close relationship to logic programming languages is a promising starting point.

Ontology Modeling Language

The *Web Ontology Language* (OWL)—standardized by the World Wide Web Consortium (W3C)[1] —is a Description Logic-based (DL) ontology language [7]. Ontologies are used for describing terms and their relationships. In OWL this is done by using a textual language based on the Resource Description Framework (RDF), which requires from users to recognize and remember a multitude of relationships provided in a sequential order. Brockmans et. al [8] have introduced the Ontology Definition Metamodel (ODM) being used as modeling language for defining ontologies—the visual modeling capabilities are introduced by defining a UML profile [40] for ODM. Even OWL rules have been integrated in an advanced ODM version [7]—the

[1] OWL information: `http://www.w3.org/2004/OWL/`, last accessed: February 24, 2011

visual representation of such rules is an important aspect for deducting solution ideas for ASP rule visualizations as well.



$$\mathrm{BadVintager}(x) \leftarrow \mathrm{ownsWinery}(x, y) \wedge \mathrm{dislikesWine}(x, z) \wedge \mathrm{hasMaker}(z, y)$$

Figure 3.4: Visualization of a OWL description logic example using rules

In the example shown in Figure 3.4 of [7] the DL rule

$$badVintager(x) \leftarrow ownsWinery(x, y) \wedge dislikesWine(x, z) \wedge hasMaker(z, y)$$

is visualized with the ODM UML Profile. The visualization is built around the three used variables *x*, *y*, and z which are placed in boxes representing the rule head or the rule body respectively. Each literal is represented as directed edge between two variables or as attribute of a variable for unary literals. This is a simple mechanism for visualizing the most important elements involved in the rule. However, this approach is limited to a maximum literal arity of two, although DL can support any arity. Literals with higher arity could be supported by introducing more-way edges—the sequence of variables in more-way edges needs an additional specification in the diagram. Another drawback is the weak binding between variables in the rule head and the rule body which is only textually listed. The rule body is connected by a directed dashed edge with the rule head.

Logic-Based Languages

For efficiently supporting the program conception process of logic-based languages, a simplification of inference relationships is necessary. This is achieved by the *Diagrammatic Predicate Logic* (DPL) which tries to model predicate logic formulas in a UML style. Solutions for predicate logic highly interrelate with ASP which is itself based on predicate logic. Another interesting approach is given which focusses on visualizing ASP programs as services rather than as code elements. As such a focussing on the program behavior can reduce the visualization complexity, this approach is chosen for a deeper inspection.

Diagrammatic Predicate Logic

The modeling of first order predicate logic (PL1) is illustrated with the assistance of the approach by Lamo [30]. In this paper, a modeling language named DPL is proposed that allows the modeling of PL1 expressions on meta and instance level. PL1 allows the definition of formulas

based on a given set of constants, function symbols, predicates, and variables—as introduced for ASP. Generally, it has to be noted that the language of ASP is a subset of PL (except default-negations) and, therefore, PL approaches of any order strongly relate to ASP.

DPL allows the definition and usage of binary relationships between predicates, e.g., inequality relationships triggered by unique key attribute differentiations among predicates. General relationship aspects are modeled in a schema, which is named the "specification"——an example is given in Figure 3.5). Relationships between particular instances of a given predicate are modeled in another diagram type, which is named "Instances of Diagrammatic Specification", e.g., *Package:p1* holding the classes *Class:c1* and *Class:c2* in the example of Figure 3.6.

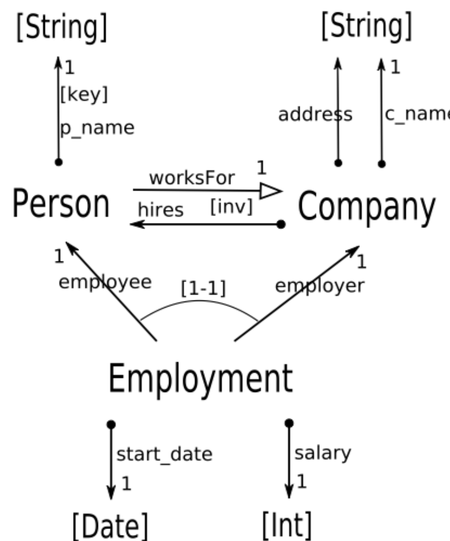


Figure 3.5: A specification diagram of PL1 [30]

In the example of Figure 3.5, a formula with the predicates *Person* and *Company* is modeled. The used arrows represent relationships. For example, the arrow from *Company* to *Person* represents a relationship for hiring employees. These arrows are labeled and are, furthermore, enhanced by defined constraints—e.g., the invariant $[Inv]$.

In Figure 3.6, the same constellation is shown on instance level. These instances are modeled in the same notation as the specification. However, no constraints are necessary to be visualized on this level. For example the $Company\ hires$ the $Person\ p2$. This relationship is, furthermore, expressed by the $Employment$ predicate.

Conceptually DPL is based on "Generalized Sketches" which represents "a graphical representation of category theory" [30]. DPL attempts to transfer this approach to the Model Driven Engineering field. Therefore, it tries to formulate the transferring from PL1 to models by defining the possible signature as "underlying graph" [30].

Disadvantages of DPL can be seen in (i) the low level of abstraction and (ii) the inability for expressing all types and variants of PL1 formulas. (i) is expressed by the focus on instance modeling rather than schema modeling. High levels of abstraction are valuable in the modeling context as it allows to describe complex systems. (ii) is related to the fact that only some very

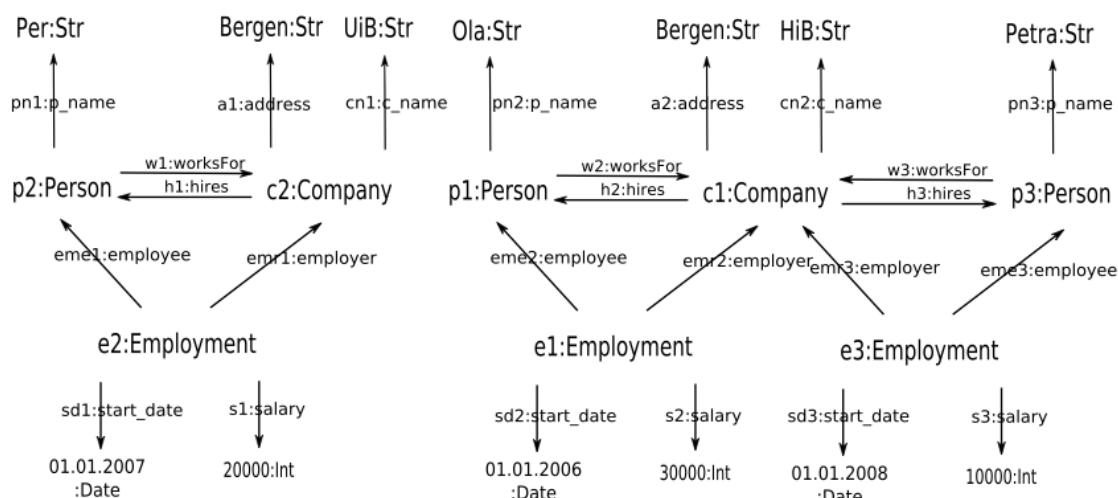


Figure 3.6: An instance specification diagram of PL1 [30]

specific scenarios—e.g., key based equality—are presented with DPL, but the building of full formulas including connectives (and, or, xor, implications), braces as mechanism for nesting terms and rich sets of used variables and parameters are left unaddressed by this approach. Furthermore, the notation does not highlight possible ways of solving, modifying, optimizing, or reducing formulas in any way. It is unclear, if this diagram can be reconverted to PL1 formulas or if PL1 formulas could be automatically visualized in this manner.

Service-Oriented Modeling with Answer Set Programming

The specification of services with preferences as models (based on a metamodel) is proposed by Confalonieri et. al [11]. From such models model transformations transfer the stored content to models conforming to the metamodel of answer set programs "with possibilistic ordered disjunctions" [11]. The stated services are accumulations of preferences referring to a certain functional—holding inputs with preconditions and outputs with postconditions—and non-functional properties. These preferences are associated with a certain goal specified by the user. All functional properties are defined to be related to services. These services are set in the environment of the *service oriented architecture* (SOA) [46].

This approach allows the definition of specific answer set programs by creating services within models. It is, therefore, directly related to a specific usage scenario. Additionally, [11] represents the first discussed approach intensively broaching the overcoming of the code-centric program development by proposing a service-centric alternative focus. Technically, these models are realized by introducing a metamodel as modeling language allowing the specification of model instances describing particular services.

However, the approach is context-dependent (dependent on SOA) as it is not trimmed to support a general design-centric development and analysis of program results. For such a design-centricity the models—conforming to the proposed metamodels—have to focus on the graphical

development proceeding in lieu of the data encapsulation (data-centricity) as it is done in [11]. Moreover, a drawback is the used specific language variant which disallows user defined language or language element variations according to technological boundaries, e.g. used solvers.

3.3 Debugging

Another relevant perspective is the usage of textual language visualizations for debugging purposes. Debugging focuses on identifying the error origins which provides an essential assistance why a certain erroneous behavior has been returned. The success of visual debugging mechanisms deeply relates to their problem-specific adaption. The more the program semantics is related to the visualization methodology, the better key elements can be highlighted. Another important aspect is the stepwise visualization of the operational semantics.

Educational Debugging

Neumerkel et. al [36] highlights the applicability of visualizations for logic programs for educational purposes. They considers visualizations as "powerful aid for learning a programming language" [36] which can be applied for logic programming as well. The approach of Neumerkel et al. enhances the GUPU (Gesprächsunterstützende Programmierübungsumgebung) programming environment—with the purpose to support students in designing and debugging Prolog [23] programs—with visualization capabilities. GUPU allows the compiling and execution of Prolog programs and assists them during the developing process. If a compilation fails, error messages are added to the program code as comment. GUPU even provides assertions like the identification of duplicate code. In addition, customized assertions can be added or even be complemented by using assertion queries. This non-graphical development assistance was enriched by *viewers* which concentrate on the elementary visualization of answer substitutions. Other aspects than answer substitutions are out of the focus of this visualization approach. Furthermore, there exist predefined "problem specific viewers" [36] which visualize answer substitutions according their context-specific semantics. For example a provided viewer [36] visualizes answer substitutions of railway networks. This "problem specific viewer" [36] visualizes all applicable cities on a map. Cities are connected by a line, whenever there exists a direct train connection. This visualization mechanism is very simple and allows the recognition of possible railway routes.

Although this pluggable context-specific visualization approach is able to provide stunning visualization results for logic programs, three drawbacks have to be considered. First, it represents a static visualization solution which does not allow any interactions except code modifications. Second, problems for which no "problem specific viewer" [36] exists, have to be visualized in an elementary viewer. Third, the operational semantics and program structures which are responsible for producing the output (e.g., rules, predicates, and variables) are not involved in the visualization—thus, no additional assistance is provided.

Model Transformation Debugging

Model transformations are typically very complex and for this purpose *declarative model transformation languages* are advancing. For such languages, however, other debugging mechanisms are necessary. Therefore, two alternative solutions for the debugging of the declarative modeling transformation language QVT-Relations (QVT-R) are briefly inspected: (i) de Lara et al. [12] and (ii) TROPIC [43]. Both approaches address the problem of debugging needs by applying the straightforward Petri net visualization. However, (ii) aims at visualizing the hidden operational semantics of declarative statements by building a flow of atomic transformations from one item to another. Therefore, it does not only visualize QVT-R, but it also offers good analysis opportunities as typical mistakes can be realized by wrong arrow linkings, missing arrows, wrong instances or similar errors [29]. In contrast (i) concentrates on visualizing abstract relationships extended by a textual notation. This visualization does not allow a stepwise inspection of the hidden operational semantics, but a much more compact way of visualizing existing relationships between elements of models.

As all declarative languages hide the operational semantics, which aggravates the ease of debugging, the approach (ii) can be transferred to other declarative languages, e.g., ASP, as well. This can be beneficial as this approach provides clear visual sequences of actions and allow the stepwise debugging of programs. However, it has to be stated that the visualization technique of (ii) tends to grow disproportionately to the involved program statements which is a drawback for more complex problems. Additionally, (i) does not support the stepwise inspection of the operational behavior of a model transformation execution.

Translational Debugging

There exist many functional programming languages like Haskell[2] or TOY [1] which have a huge community behind them. In lieu of states, variable assignments, and defined commands as known from imperative programming languages, functional languages only make use of (mathematical) functions.

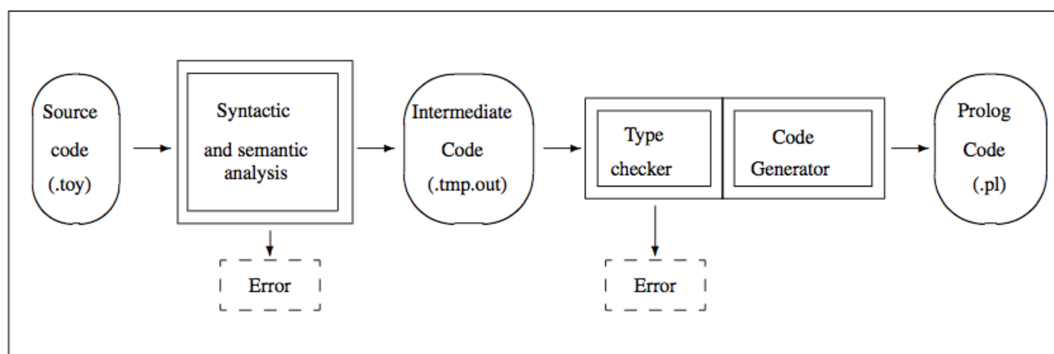


Figure 3.7: Transformation of a TOY code file to a Prolog representation [9]

[2]Haskell information and download: `http://haskell.org/`, last accessed: February 24, 2011

A translational approach for debugging of such languages is the "graphical declarative debugger of incorrect answers for the constraint lazy functional-logic programming language TOY" [9]. The debugging in a step-by-step manner introduced and used for many imperative languages is often "not suitable for debugging declarative programming languages" [9]. However, this debugger is concentrates on the identification of errors comprised in programs written in functional languages like TOY. As visualized in Figure 3.7 the debugging process starts with the raw TOY code file as input. This file is syntactically and semantically analyzed and transformed to an intermediary code file. The resulting code is again checked by a type checker and afterwards Prolog code is generated. The last step is undertaken because this debugger is written in Prolog. This debugger then obtains a formal computation tree that is the entry point for debugging. From this point on the developer can freely inspect the results of the computation tree or call provided strategies "or finding out a buggy node and hence an incorrect program rule" [9]. Furthermore, it obviously holds that TOY can be used for debugging Prolog code as well. The computation from TOY to Prolog is a translational strategy that tries to bring the language on a better analyzable platform.

Translational approaches provide the possibility of profiting from other languages, platforms, or representations. However, the identification reasons for problems requires two transformation steps. First, an initial forward transformation—allowing the debugging—and second the backward transformation of the identified error causes to its original representation is necessary. This reduces the debugging dynamics and increases the risk of transformation errors. However, such solutions are appropriate, if native solutions are too effortful or cannot be established.

3.4 Analysis & Verification

Besides the approaches applicable for initial program designs and concepts usable for identifying particular error causes, techniques are necessary which allow proofing, verifying, and inspecting of existing programs. To verify or analyze the appropriateness of a solution, it is essential to (a) make the effects of code element dependencies, e.g., rules, and (b) the program structures explicit. Aspect (a) is taken on by using dependency graphs highlighting applicable sequences of predicate usages. Aspect (b) can be interpreted in two ways as in ASP programs rules and variables have a strong effect on the overall program behavior. Variable dependencies, equalities and inequalities are tackled by *String Diagrams*, whereas rule distances—giving an idea of applicable sequences of rule applications—are visualized by *Colored Graphs*. If none of these two perspectives provide appropriate development support, they may be assisted by an approach based on the *Human Usable Textual Notation* (HUTN)—originally intended for visualizing models—which allows the semantic organization of code blocks.

Dependency Graph

Sureshkumar et al. [48] states that effective tool support provided by IDEs is crucial for the success of ASP. This statement results from a survey among ASP developers that should identify

the most essential IDE features for developing ASPs. It is concluded that a dependency graph has "demonstrated strong support" [48] and was, therefore, implemented.

$$\Pi_{3.1} = \begin{cases} r1: \{\ \textit{martian(P)}, \textit{venetian(P)}\ \}\ \textit{1} & \leftarrow \textit{person(P).} \\ r2: \{\textit{female(P), male(P)}\ \}\ \textit{1} & \leftarrow \textit{person(P).} \\ r3: \textit{lies(P)} & \leftarrow \textit{person(P), martian(P), female(P).} \\ r4: \textit{lies(P)} & \leftarrow \textit{person(P), venetian(P), male(P).} \\ r5: \textit{truthful(P)} & \leftarrow \textit{person(P), martian(P), male(P).} \\ r6: \textit{truthful(P)} & \leftarrow \textit{person(P), venetian(P), female(P).} \\ r7: & \leftarrow \textit{person(P), lies(P), truthful(P).} \end{cases} \tag{3.1}$$

The dependency graph shown in Figure 3.8 visualizes the dependencies of predicates used in an example program $\Pi_{3.1}$. The presented approach does not make use of any form of negation, epistemic or, or any instance information such as Π. It is, however, a very useful and simple demonstration of how the literals stated in $\Pi_{3.1}$ interrelate. As it does not focus on facts and other language elements it cannot be used for explaining a particular result.

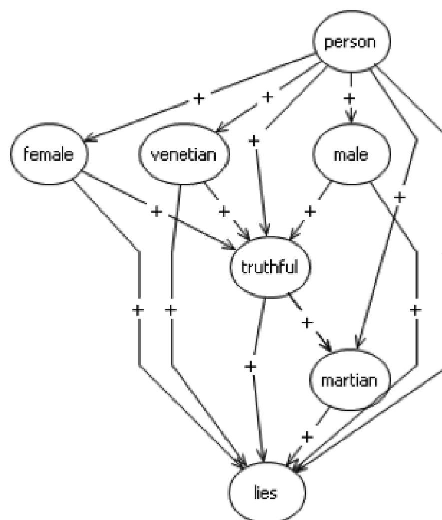


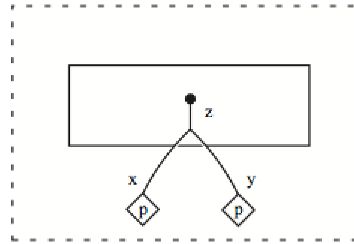
Figure 3.8: A dependency graph of a simple answer set program [48]

The graph of Figure 3.8 makes only use of a single variable *X*. The predicates, therefore, are in the form of *person(X)* or *female(X)*. The stated approach does not discuss how different variables or constants could be integrated in the dependency graph. It is, however, very likely that several graphs could result from a single program in more sophisticated cases. Dependency graphs, therefore, represent a good visualization basement which, however, has to be extended to support a set relevant missing program elements, e.g., rule dependencies, higher arities, or facts.

String Diagram

A diagrammatic approach for assisting the PL1 verification is provided by [6]. It aims to transfer each PL1 theory to Pierce systems (monoidal 2-category of relations) for which Pierce himself provides a "graphical system for handling first-order calculus of relations". This is modernized in [6] by transferring it to categorical and geometrical logics. The geometrical visualization is realized by using so called *String Diagrams* [6] which are enhanced for this application.

This approach allows a good recognition of equalities and inequalities of used variables. The full visualization focuses on the description of variables, their interrelations, and possible equalities with other variables. Such diagrams can be rewritten to step-by-step transform a formula in equivalent representations—allowing graphical proofs of formula equivalences. In [6], it is claimed that several proofs could be found, with the help of the stated approach. The rewriting is formally defined by using deformation rules. The rewriting typically involves several steps, which can be aggregated as well.



$$\exists x \exists y \neg(\exists z[z = x \wedge z = y]) \wedge p(x) \wedge p(y).$$

Figure 3.9: A diagrammatic representation of a simple PL1 formula as *String Diagram* [6]

In Figure 3.9, an example for visualizing simple PL1 formulas is shown. The diamonds—$p(X)$ and $p(Y)$—are symbols for predicates. These predicates are bound to variables by using edges labelled by the name of these variables—e.g., to X. Variables used within the same boxed structure could be, but need not be, equal. As this formula (that is stated below the diagrammatic representation) ensures the inequality between X and Z, as well as between Y and Z, the variable Z is placed within an own boxed structure. For ensuring inequality for all three variables all three would have to be placed in different structures—which would, therefore, lead to at least two boxes within the dashed environment.

This approach is very useful for finding proofs and reductions of formulas by using eliminations and deformations. As the graphical representation is mainly based on descriptions of variables in lieu of focusing on formulas itself (leaving some PL1 concepts behind), this approach is not intended to be used for the design of PL1 formulas.

Colored Graph

Colored graphs of Konczak et al. [28] for ASP programs focus on the visualization of dependencies between rules in Colored Graphs. This is therefore intended to provide an assistance in recognizing why rules can or cannot be applied. The applicability of rules in a given state (an answer set being constructed for a program Π) is visualized by colorings of edges. It is, therefore, also possible to provide graphs representing operational information.

$$\Pi_{3.2} = \begin{cases} r1 : p & \leftarrow . \\ r2 : b & \leftarrow p. \\ r3 : f & \leftarrow b, \text{not } f'. \\ r4 : f' & \leftarrow p, \text{not } f. \\ r5 : b & \leftarrow m. \\ r6 : x & \leftarrow f, f', \text{not } x. \end{cases} \tag{3.2}$$

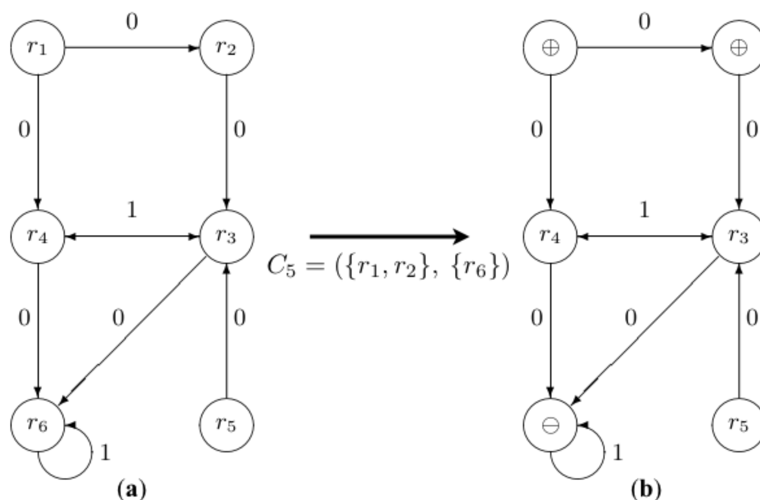


Figure 3.10: Colored graphs of an answer set program [28]

An exemplary program shown in Equation 3.2 is visualized as colored dependency graphs in Figure 3.10. The graph highlights the distance between particular rules—the steps which are necessary to be apply a rule as a consequence of another rule. For example in (a) the rule $r1$ of $\Pi_{3.2}$ has a directed edge to $r2$. This edge is labelled with the distance of 0. This distance describes the reachability of $r2$ from $r1$. As the result (head) of $r1$ represents the body literal of $r2$, the rule $r2$ can directly be applied after $r1$ can be applied—therefore, the distance is 0. In other notations the distance of rules defines the color of edges—green, blue, and red. In *(b)* of Figure 3.10 the coloring C_5 is added to the graph which represents some instance information (facts). This coloring marks involved rules as reachable (⊕) or unreachable states (⊖). In *(b)*

still rules $r1$ and $r2$ of (a) are reachable. This is not the case for $r6$ which is labeled by $\ominus$. This results from the definition of $r6$ that states that if no evidence of x (*not* x) exists, x is returned. After applying this rule once, it cannot be applied a second time. The coloring C_1 already holds $r6$ and, therefore, is unreachable using all edges. The unary edge from $r6$ to $r6$ holds the distance of 1 in both *(a)* and *(b)* as it cannot be directly applied after $r6$ has been called again. A special case is the edge between $r3$ and $r4$ which is bidirectional with a distance of 1 as it only allows the application one of both rules.

This approach is useful for identifying the applicability of rules in a certain program state. However, it cannot express the relationships of variables, the usage of constants and, therefore, the reasoning for the outcome (certain answer sets). The notation, however, is very compact and the approach using distances between rules provides precise way of describing relationships.

HUTN

HUTN [38]—which was standardized by OMG[3]—is an approach which focuses on visually and meaningfully organizing textual code elements in recognizable blocks. Its suitability for representing declarative languages is shown by Pau Giner et al. [20] who have introduced a Test-Driven-Development concept for transformation languages. In particular, the HUTN notation is used for organizing properties related to the testing of transformation languages. It is furthermore stated by [20] that HUTN is generic and could be "applied to any MOF-based meta-model" and would be fully automated and "no human intervention" would be necessary. As it is named "Human Usable", HUTN is designed to be human-understandable and easy modifiable.

<table>
<tr><td>Num</td><td>Version</td><td>Intent</td><td>Mapping: UML2DB</td></tr>
<tr><td>1</td><td>1</td><td colspan="2">A concrete class generates a table with the same name.</td></tr>
<tr><td colspan="2">Test data</td><td colspan="2">Expected result</td></tr>
<tr><td colspan="2">Class "Class1"{
isAbstract: false
}</td><td colspan="2">Inclusion: Table "Class1"{
columns: Column "PK_Class1"{
}
}
Assertion: "Has primary key"
self.getPrimaryKey().isDefined()
Assertion: "PK is the key column"
self.getPrimaryKey().members.first() =
self.columns->any(c|c.name='PK_Class1')</td></tr>
</table>

Figure 3.11: A HUTN example representation [20]

In Figure 3.11, there exists a class *Class1* in the section *Test Data*. This information is provided as input for the test case. On the right side there is placed the *Expected result*. It is mainly dividable in "result parts and assertions" [20]. In the result part the expected result can be defined to be "inclusion, exclusion or exact" [20]. The assertions are named OCL-queries, which have to return true. If the expected result for a specified input matches the outcome and all assertions return true, the test case is correct.

[3] HUTN information: `http://www.omg.org/spec/HUTN/`, last accessed: February 24, 2011

For the context of ASP the capability of organizing textual in visual and semantically organized groups, seems to be a transferrable assistance in designing ASP programs. However, the complexity of ASP rules and their dependencies is often so high that a code organization might not be an applicable exclusive solution as development support.

3.5 Summary

In this section, several approaches for visual development support for different declarative languages were presented. The most directly applicable approaches for ASP are dependency graphs and Colored Graphs. Dependency graphs allow the visualization of hierarchical literal dependencies without involving particular rules nor variables and constants. Colored graphs in contrast are able to visualize the rule interrelations by highlighting the distances between rules. This distance is useful for identifying which rule can be applied in which state (after executing a certain other rule). To be able to identify the behaviour of answer set programs, it is necessary to understand the rule interrelations and the variable interactions. Such variable interactions are not used by any discussed answer set program visualization except of String Diagrams, which describe argument dependencies for predicate logic in a compact way. An extensive and service-specific approach is presented by [11] where a service-model is specified and afterwards transferred to an ASP model. This concentration on the program behavior rather than on visualizing syntactic code elements, can reduce the solution complexity. However, the approach is hampered by its specific nature and its data-centric models. Nevertheless, these four approaches are highly interesting for the future visualization of answer set programs. Additionally, the simplicity of the Diagrammatic Predicate Logic is a useful example for logics visualization.

However, none of the mentioned seems applicable for completely supporting the software development process. On the one hand, no single concept was available assisting in the conception, the debugging, and the verification of ASP programs at the same time. On the other hand, no solution could be identified which is capable of abstractly visualizing all essential elements or even comprised deductions of ASP programs. For this purpose, it is the aim following sections to propose more complete solutions being optimized for ASP.

CHAPTER 4

Visualization of Non-Deductive Answer Set Programs

In this section, an initial approach for visualizing some aspects of ASP programs as a first conceptual visualization attempt is given, which mainly tackles the program conception task. This initial solution is based on *Entity-Relationship (ER) diagrams* which are well established in database modeling field. Their strength is their capability of describing relationships between database tables and structures. For the context of ASP such a visualization can be used for describing structural and static relationships of a program Π. Elements of Π are facts, constraints, and other rules (deductive rules). Facts and rules are known elements of classical database and are, therefore, describable in terms of ER-diagrams. This idea is used to undertake a first visualization attempt which focus on these two rule types. The restriction to facts and constraints is removed in consecutive approaches found in the next sections.

4.1 The Big Picture

Contrary to the introduced related works, the proposed approach should be applicable for supporting the program conception. This is achieved by assisting in the design process of the predicate formats—the predicate name and the sequence of arguments of a predicate—extractable from literals used in ASP programs. The predicate format is dependent on related values (attributes), relationships to other predicates (explicit constraints) and general design decisions (implicit constraints), e.g., primary keys. However, the textual modeling of relationships is often effortful and intransparent. For this purpose, a concept based on ER-diagrams is proposed to provide an abstract structural design layer above textual decisions. This abstract design layer is capable of structurally and parallel visualizing a multitude of relationships and their comprised properties.

The resulting ER-diagram of this abstract design layer defines a schema transferred to an ASP program whose basis is a created tool for generating facts. This initial program is non-

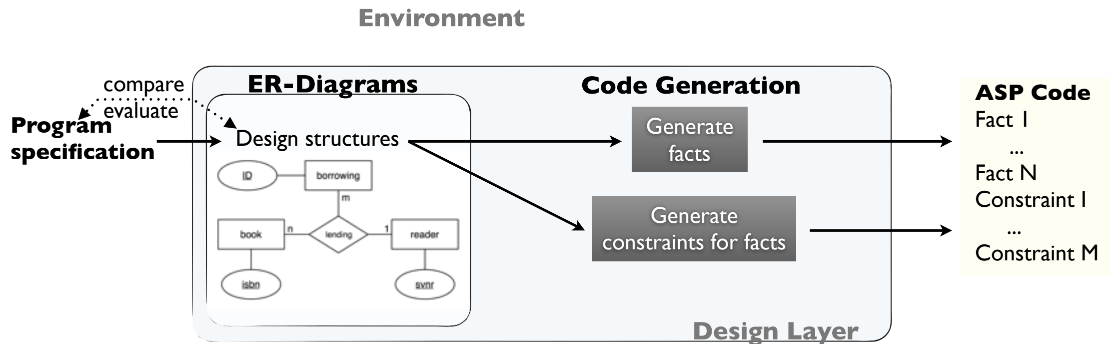


Figure 4.1: The big picture of the non-deductive ASP program visualization and its environment

deductive as it only comprises facts as well as constraints. As the derived predicate format is heteronomous, assistance in defining particular facts for the modeled ASP program is essential. This is undertaken in a final generation step.

The big picture of the discussed approach is shown in Figure 4.1—grey boxes indicate the technical assistance provided by the introduced approach. For a given *program specification*, the abstract *Design Layer* of this approach is used for designing non-deductive ASP programs. The ER-diagrams are used for designing the structures of ASP programs—predicate formats based on predicates and their arguments. Thereafter, the *supporting technologies* can be used for *generating facts* and for *generating constraints* for these facts. Beyond the boundaries of this layer, classical methods can be used as well for comparing and evaluating the generated code against the original specification or for enhancing it with deductive rules.

4.2 Describing Non-Deductive Answer Set Programs

In this section, a bridge between ER-based visualization and ASP code is constructed by discussing the mapping of facts, their relationship, and constraints. The textual representation of a predicate including all of its arguments, is called the *predicate format* in this thesis. The predicate format—which is the basement for other definitions—is implicitly discussed by proposing the fact mappings.

Entities

An entity in an ER-diagram represents a database table. Each entity can hold a set of attributes referring to columns of tables holding primitive type information. A primary key is used to ensure the uniqueness of entity instances—primary keys are marked in the diagram with a text-decoration.

These entities with their attributes can be transferred to ASP entities. Such ASP entities are atoms of the visualized entities in a textual ASP code representation. The definition of entities implicitly extends the set of available predicates which are the basis for textually specified or

generated literals of a program's rules. These predicates hold a certain textual sequence of arguments which is named *predicate format* in this thesis.

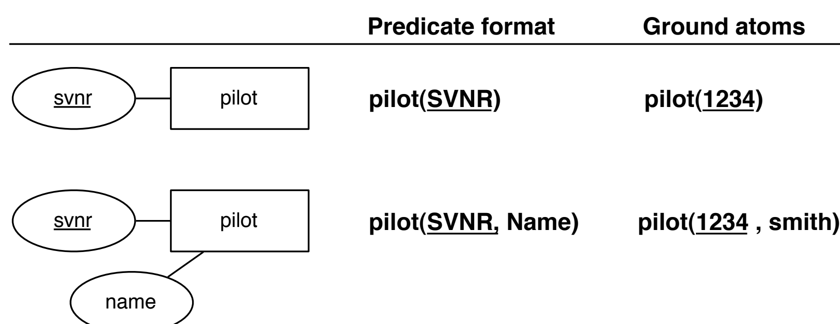


Figure 4.2: The transformation of ER-diagram tables and attributes to ASP code

Each attribute—including primary keys—of an entity is transferred to an argument of the textual representation. In Figure 4.2 two examples are given which highlight the basic entity transformation. The primary key *SVNR* and the attribute *Name* represent arguments of the predicate *pilot*—cf. central column of Figure 4.2 using variables to demonstrate the predicate format. Exemplary ground atoms for the predicate *pilot*—e.g., *pilot(1234, smith)* (the text-decoration is only used to highlight the transformation process)—are given on the right column.

Relationships

The most commonly used relationship variant in ER-diagrams is the binary relationship (cf. Figure 4.4). It involves exactly two referenced entities and can be considered to be the most common relationship type. Additionally, unary relationship exist which represent relationships from an entity to itself. Another applicable ER-diagram relationship is the so called n-ary relationship (cf. Figure 4.3). Not all n-ary relationships are sufficiently replaceable by binary relationships and, therefore, need an explicit integration in the visualization of the *design layer*. For solvers supporting cardinalities—lower and upper bounds of referred instances—further considerations are necessary. In particular, a differentiation in (i) *one-to-one* (1:1), (ii) *one-to-many* (1:n), and (iii) *many-to-many* (n:m) relationship variants is required (cf. Figure 4.4).

(i) One-to-one related entities are in a relationship between two entities, where exactly one instance of each entity exists. Consequently, none of these entities directly depends on the other entity—e.g., a *person* can reference an *airport*, but an *airport* can also reference a *person* (cf. Figure 4.4). (ii) One-to-many related entities express a relationship from one entity instance to a set of instances of another entity. (iii) In one-to-many relationships, each instance of one entity can refer to a number of instances of another entity and vice versa. For this purpose, such relationships represent own database tables (in the background) which store the references between these entities.

To transform such relationships to textual ASP code, the differentiation according the cardinalities of the relationships is most essential (cf. Figure 4.5). (i) As one-to-one relationships

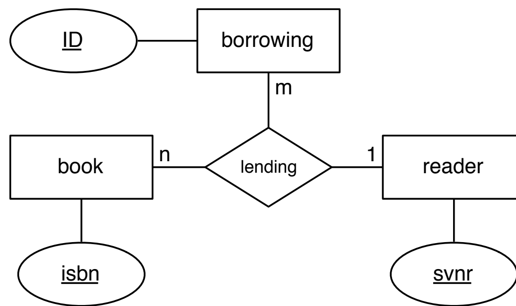


Figure 4.3: Ternary relationship in ER-diagrams

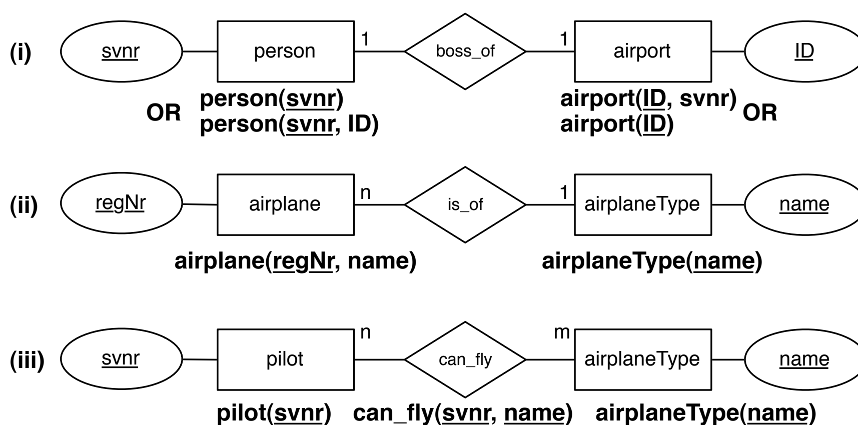


Figure 4.4: The cardinalities of relationships of ER-diagrams

imply a direct relationship between two single entity instances, the primary key of one entity is added as argument (foreign key) to the second predicate. For unary relationships (an entity pointing to itself) a single solution is available, whereas for binary two alternative transformations exist, which are syntactically equivalent, but can provide program-specific optimizations (e.g., an *engine* logically depends to a *vehicle* rather than vice versa). (ii) In cases of one-to-many related predicates, the key of the predicate being targeted by an edge to this relationship with a cardinality of one (one-related predicate) is added to the other predicate (many-related) as foreign key. For relationships which are not of binary nature (involving exactly two predicates) an additional textual predicate is required for expressing the relationship. (iii) All many-to-many relationships require an additional implicit predicate representing the relationship. All primary keys of all referenced predicates are added as attributes of this additional predicate. They together represent a joined primary key (cf. Figure 4.5).

Ternary relationships, a special case of n-ary relationships, have to be transformed differently. Each relationship is textual described by an implicit entity holding the keys of each involved other entity. The primary key is represented by a joined key of all keys representing

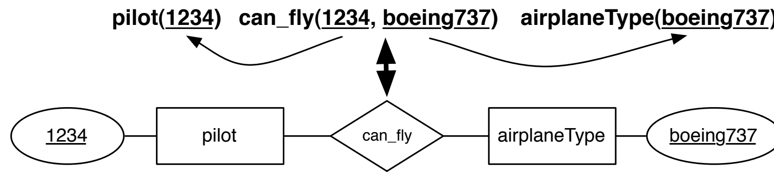


Figure 4.5: Transferring predicates to an intermediary representation

predicates being targeted by this relationship with a cardinality greater than one. These keys require an identical handling as proposed for binary relationships.

Constraints

Two types of ASP constraints can be derived from ER-diagrams. On the one hand, primary key constraints which require the uniqueness of an identifier for predicates. For this purpose, constraint rules are necessary which are violated whenever two atoms with the same primary key, but different values of other attributes, exist (cf. Figure 4.6), e.g., two *airports* with the same *ID*, but a different *capacity*, exist. On the other hand, each argument of a predicate represents a certain (primitive or complex) type. For this purpose, the assignments for a predicate have to be checked for their type consistency. In addition, cardinality constraints are already restricted by the definition of a number of arguments for a predicate. Cardinality constraints, therefore, represent implicit constraints whereas all others are explicit.

4.3 Code Generation

The code generation process starts by identifying the textual predicate format which is based on the used attributes and relationships to other predicates. On this basis, general constraints are generated and afterwards supplemented by the generation of particular facts based on tuples of constants. The code generator is intended to support the syntax of the DLV solver.

Generation of Constraints

Primary key constraints. To validate if a primary key constraint is violated, a set of constraint rules is necessary for a single primary key. Each constraint rule validates, if for one primary key two different tuples exist in the same candidate answer set—an answer set not validated against the constraint rules of the program. So, for for each non-primary key attribute a constraint rule is necessary which is violated whenever at minimum two different values for the same primary key exist. The following example shows the uniqueness validation of the primary key *R1* for *airplane*. The constraint itself is formulated in a second rule to improve the ease of debugging whenever a constraint is violated.

```
nok_pk_airplane_type(R1) :- airplane(R1, T1, _),
airplane(R1, T2, _), T1 != T2.
:- nok_pk_airplane_type(R1), airplane(R1, _, _).
```

Type constraints. The type constraints are generated by identifying if the arguments of a predicate represent an attribute or a foreign key—an attribute referring to another entity. It has to be avoided that values not conforming to the type of the attribute can be used as attribute. If an argument represents a foreign key, it has to be ensured that this value is used as primary key of an instance of this referred entity. For these purposes, type constraints have to be generated. As literals using free variables and *default*-negation would be necessary to express such constraints, the usage of a single rule would be unsafe in DLV. For this purpose, all constraints are represented by two rules in the following manner.

```
ok_airPlane_type(R1) :- airplane(R1, T1, _), airplaneType(T1, _).
:- not ok_airPlane_type(R1), airplane(R1, _, _).
```

If an argument represents an attribute of a predicate, a fictive type is constructed named by using the attribute name plus the postfix "_val", e.g., "capacity_val".

Generation of Facts

In the previous sections, the applicable predicates and the necessary constraint rules were defined. These predicates can be used to formulate the interpretation of a program Π by adding facts holding predicates with constant values. This section is dedicated to the automated formalization of such facts from a given ER-diagram expressing Π. For each predicate a set of tuples of constants can be applied which have to express the values representing this predicate and all of its relationships. For example for the predicate *airplane* of the Figure 4.6 the tuple $\langle$*1, Boeing737, 1*$\rangle$ can be applied.

The automatization is achieved by interpreting the ER-diagram. The ER-diagram is then transferred to a set of predicates. Afterwards commands can be applied for adding the values of a predicate. In the background, they are automatically transformed to valid predicates. This is achieved by allowing the usage of commands, e.g. "add" and "format". The "add" command allows the adding of one fact. The "format" command can print the argument sequence of a predicate. Each argument representing a value of another predicate (foreign key) is displayed with the name of the predicate it depends to and the name of the argument itself. This is necessary as arguments need not have unique names over all involved predicates. Each argument, however, has to be named uniquely for one predicate.

4.4 Evaluation

In this section, the implementation is evaluated by an example. This is undertaken by proposing an exemplary non-deductive ASP program being visualized as ER-diagram which is generated to textual ASP code. This example is evaluated in several steps afterwards.

Example

An answer set program which aims at organizing the placement of *airplanes* is used to illustrated the above presented concepts (cf. Figure 4.6). Each *airplane* is assigned to an *airport* which

represents its home base. Each *airport* has a given *capacity*. Each *airplane* assigned to an *airport* requires a subset of the *airport capacity* according its own *size*. The *size* is defined by the *airplaneType* of the *airplane*. Each predicate uses a primary key, e.g., the *name* of the *airplaneType*. This scenario can be visualized as ER-diagram (cf. Figure 4.6).

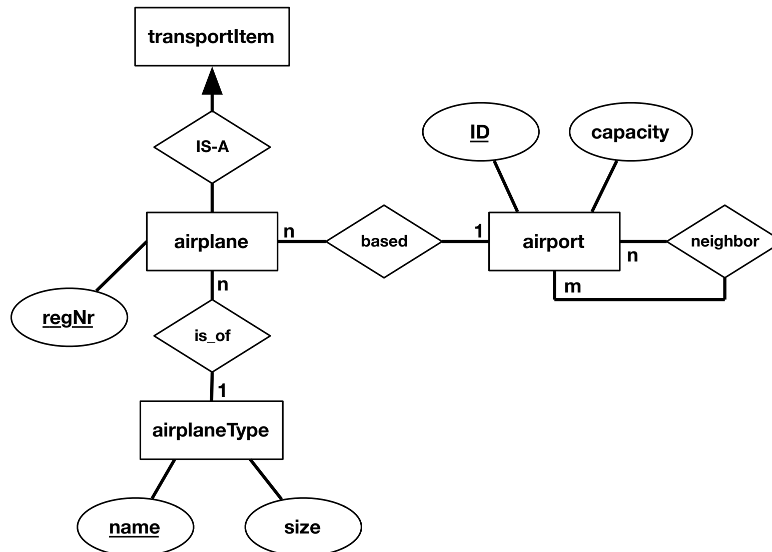


Figure 4.6: The ER-diagram representing an answer set program for the scheduling of airplanes

The corresponding ER-diagram can be transformed to the following answer set code according the DLV syntax:

```
%PRIMARY KEY CONSTRAINTS
%(each ID can only occur once)
nok_pk_airplane_type(R1) :- airplane(R1, T1, _),
       airplane(R1, T2, _), T1 != T2.
:- nok_pk_airplane_type(R1), airplane(R1, _, _).
nok_pk_airplane_base(R1) :- airplane(R1, _, HB1),
       airplane(R1, _, HB2), HB1 != HB2.
:- nok_pk_airplane_base(R1), airplane(R1, _, _).
nok_pk_airport_capacity(ID) :- airport(ID, C1), airport(ID, C2),
       C1 != C2.
:- nok_pk_airport_capacity(ID), airport(ID, _).
nok_pk_airplaneType_size(N) :- airplaneType(N, S1),
       airplaneType(N, S2), S1 != S2.
:- nok_pk_airplaneType_size(N), airplaneType(N, _).

%TYPE CONSTRAINTS
ok_airplane_type(R1) :- airplane(R1, T1, _), airplaneType(T1, _).
```

```
:- not ok_airPlane_type(R1), airplane(R1, _, _).

ok_based_type(R1) :- airplane(R1, _, HB1), airport(HB1, _).
:- not ok_base_type(R1), airplane(R1, _, _).

ok_airport_capacity_type(HB) :- airport(HB1, C1), capacity_val(C1).
:- not ok_airport_capacity_type(HB), airport(HB, _).

ok_airplaneType_size_type(T) :- airplaneType(T, S1),
        airplaneType_val(S1).
:- ok_airplaneType_size_type(T), airplaneType(T).

ok_neighbor_type(HB1, HB2) :- neighbor(HB1, HB2), airport(HB1, _),
       airport(HB2, _).
:- not ok_neighbor_type(HB1, HB2), neighbor(HB1, HB2).
```

The generated code considers the transformation of cardinalities, primary key uniqueness constraints, and type constraints. The sequence of arguments for each predicate is implicitly given by the used attributes of the entities. Furthermore, for each entity of this example a set of facts can be generated. The following example highlights this generation process by showing the usage of the *format* and *add* command. First, the sequence of the arguments of *airplane* is printed. Then an *airplane* is added as fact including an implicit displaying of the predicate format. The argument values (constants) are then entered separated by a space. To support space characters within a value block quotations marks can be used, e.g., "Ëoeing 737̈"

```
%TYPE A COMMAND
:format airplane
regNr airplaneType.name airport.ID %THE PRINTED FORMAT

:add airplane
regNr airplaneType.name airport.ID %AUTOMATICALLY PRINTED FORMAT
%USER ADDS ONE AIRPLANE WITH THE FOLLOWING VALUES
:1 Boeing737 1

%OR USE QUOTATIONS MARKS
:1 "Boeing 737" 1

%RRESULTING FACT
airplane(1, Boeing737, 1) :- .
```

Lessons Learned

The approach using ER-diagrams for visualizing answer set programs is able to express cardinality and type constraints. Such constraints are typically very verbose in textual representations and, therefore, represent a strong asset for designing ASP programs. An additional strength is the representation of inheritance relationships (cf. Figure 4.7) in ER-diagrams—they could be integrated in future approaches—which allow the simplification of typical ASP problems. For example, *birds* can have the optional capability to fly. A differentiation of subtypes of birds according their capabilities can be expressed by using an inheritance relationship. In particular, two subtypes of *bird* are shown in Figure 4.7: *non_flying_bird* and *flying_bird*. From these subclasses a lot of animals can inherit, e.g., a *penguin* has to be a *non_flying_bird*. Often such patterns are more difficult expressible in textual representations then in ER-diagrams and, therefore, simplify the design process of such constellations.

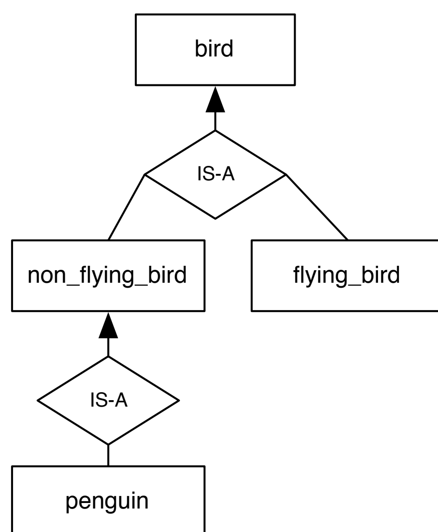


Figure 4.7: Inheritance patterns in answer set programs visualized with an ER-diagram

As a consequence, an important result of this initial approach is the necessity of visualizing relationships compromised in ASP programs. Even the integration of inheritance relationship, ternary relationships and further structural dependencies could increase the expressing power of the given approach.

The most essential learned lesson is the power of simplified visualizations allowing the paralleling of relationship representations. Often long lists of constraint rules are visualizable in a small ER-diagrams (cf. Figure 4.6). In addition, the automated fact generation—which is textually complex for predicates being involved in a set of relationships—strongly decreases the problems of wrongly specifying values for arguments of literals being dependent on other definitions. For this purpose, concepts for fact specifications should be integrated in future solutions as well. Moreover, such ER-diagrams are well-known and, therefore, reduce the effort in initially understanding this visualization technique.

Furthermore, it can be stated that the proposed approach is applicable for assisting in the program conception, debugging, and verification task of the ASP development. The initial program conception is a strong asset of ER-diagrams which are intended for being used for initial visualizations of structures. Additionally, typical pitfalls such as the missing of a relationship—often hardly recognizable in textual notations—and the wrong specification of types—e.g., primitive type attribute instead of an intended relationship to another complex type—are easily recognizable. This simplifies the debugging process for non-deductive ASP programs. Moreover, the verification of ASP programs is supported as well by allowing comparing the visual diagram elements with the specification. This process could be automated for typical models and formalized program specifications.

4.5 Critical Discussion

Beyond these boundaries of non-deductive programs functional requirements in entire deductive programs exist which cannot be visualized in such a manner. In particular, inferences cannot be described with ER-diagrams, e.g., a constraint for validating that the *capacity* of *airports* is not exceeded (cf. Figure 4.6), cannot be derived from this visualization. As a consequence, the described solution is only applicable for modeling the data basis of ASP programs. Additionally, it has to be noted that facts are not directly integrated in the ER-diagrams which would represent a useful future extension. From these experiences in visualizing non-deductive ASP program, an approach for deductive programs is established in forthcoming sections which tries to remove the described limitations.

CHAPTER 5

Visualizing Inferences & Design Decisions

The approach proposed in Section 4 realizes a simple method for visualizing ASP programs by reusing concepts of ER-diagrams. However, this approach only considered a subset of the ASP language. Powerful features like inferences, negations, or disjunctions were neglected in favor of reducing the visualization complexity. In this section, we therefore introduce a graphical approach for answer set programs based on [16] and inspired by the concepts of Section 3 providing such features as extensions. The proposed approach is established by developing a first solution for propositional answer set programs. Thereafter, the visualization is stepwise improved to finally support answer set programs of arbitrary arity. This proceeding allows an iterative enhancement and integration of simple concepts.

The new approach is established by combining several beneficial concepts. In particular, the visualization simplicity, the clear defined abstraction levels, and the good technical support are realized by the usage of Model Driven Engineering[1] (MDE) [45] techniques.

The presentation of the proposed approach is, therefore structured as follows: First, a visualization approach for propositional ASP programs, i.e., programs only consisting of predicates with arity zero, is proposed. Second, the approach is extended for programs with an arity not bigger than one. Third, concepts for programs consisting of predicates with arbitrary arity are presented.

5.1 The Big Picture

Typically the design of an answer set program Π is undertaken on code basis. From a specification, a set of answer set rules is created by the developer. These rules are then executed by

[1]Most popular is the Model Driven Architecture approach proposed by the Object Management Group—Information: `http://www.omg.org/mda/`, last accessed: February 24, 2011

a solver which calculates possible answer sets. Such solvers are black boxes hiding the actual execution, i.e, the calculation of answer sets, from the user.

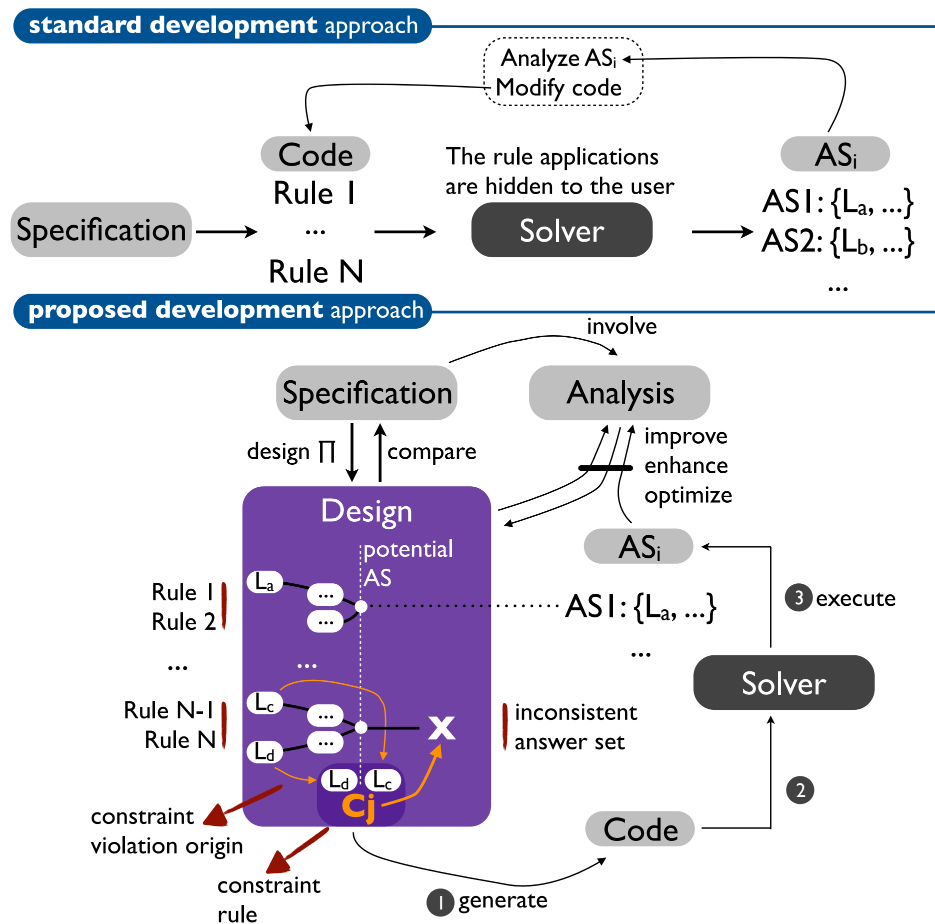


Figure 5.1: The big picture

From the textual representation of the code, inter-dependencies of the rules are not directly visualized. The reasons for answer sets, therefore, can only be identified by analyzing the returned answer sets themselves. This interpretation then allows the correction of the initially created code (cf. the *standard development approach* in Figure 5.1), if an erroneous behavior is identified. The improvement of the code—and its expected answer sets—can then be checked by re-executing the program and interpreting the changes in the returned answer sets.

To overcome this code-centricity, we establish a design layer and a design-first methodology (cf. the *proposed development approach* in Figure 5.1). This layer is responsible for visually providing assistance in analyzing the modalities of rule applications associated with a returned answer set. Furthermore, it may be essential to provide a basis for the assistance in identifying causes for unexpected results, i.e., an explanation why an expected answer set is not returned or an unexpected answer set is returned. This is done by structurally visualizing the effects of

constraint rules to the construction of a program's answer sets—cf. the *inconsistent answer set* in Figure 5.1. In a first attempt direct structural relationships between possible violation origins and the violated constraints (cf. *constraint violation origin* in Figure 5.1) are established to improve the understandability of Π*'s* behavior. This allows a real-time solver-based simulation of constraint violations.

To achieve such a design-centric development, the program Π, which is based on a specification, is visualized in an abstract and graphical design model representing the possible rule applications. Our approach follows ideas similar to the non-visual concept of [16] which represents a formal meta-program of the original program expressing constraint violations. In particular [16] computes a meta program $\mathcal{D}(\Pi)$ for a program Π by projecting the relevant answer sets to the predicates of Π. Consider the following exemplary answer set $\mathtt{A} = \{ int(l_{night}), int(l_{bright}), int(l_{candlelight}), violated(l_{r3}) \}$ for program $\Pi_{5.1}$ which states that *night*, *bright*, and *candlelight* are part of one particular answer set. However, as the constraint rule *r3* eliminates expected answer sets holding *bright* and *night* without *torch_on*, a violation (*violated(...)*) for this particular expected answer set is stated. The term *int* represents the considered interpretation of Π. Consequently, $int(l_{candlelight})$ refers to the valid containment of the literal $candlelight$ in the considered interpretation.

$$\Pi_{5.1} = \begin{cases} r1 = & night \lor day \leftarrow, \\ r2 = & bright \leftarrow candlelight \\ r3 = & \leftarrow night, bright, not\ torch_on. \\ r4 = & candlelight \leftarrow . \end{cases} \tag{5.1}$$

After establishing an MDE-based design model (integrating ideas of [16]), the model serves as basis for analyzing and redesigning Π. The model can be even transformed to ASP code—cf. (1) in Figure 5.1—and thereafter be executed by a solver—cf. (2) and (3) in Figure 5.1. As a consequence, the returned answer sets provided by the solver can be involved in further analysis of Π as well. The final model, the generated code, and the returned answer sets can be used to be validated against the original specification. In this thesis, we focus on sketching the basic workflow only and discuss the other topics as possible topics of future work.

As the design layer should allow the analysis of possible rule application sequences, rules are an integral part of the graphical representation. In particular, there is a necessity for visualizing rule interrelations, and the involved literals and their conditions, i.e., the used arguments of literals.

5.2 Syntactical Preliminaries

As the presented approaches are based on graphs, the relevant graph concepts are revisited in this section and extended by definitions specific to the visualization concepts for deductive ASP programs proposed in this thesis.

Graphs Revisited

The proposed visualization is based on concepts used for Model Driven Engineering (abbreviated as model-based approach). Models in this context may be considered as graphs which provide some useful properties for the realization of our approach. This is assisted by clearly specified abstraction hierarchies [25], which provide an abstract alphabet definition for models in each degree of abstraction. Additionally, models can be cut in semantical pieces named submodels—this can reduce the visualization complexity. For these reasons, models are applied in this thesis and as a consequence, the used graph concepts are shortly revisited. In this section, graphical models are always named models—models from the ASP context are named answer sets.

Definition 16 (Directed graph) *The ordered pair* $G = \langle \mathcal{V}, \mathcal{E} \rangle$ *of the form* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$*—where* $\mathcal{V}$ *is a finite number of vertices and* $\mathcal{E}$ *a finite number of edges—is a directed graph. A graph of the form* $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *where* $\mathcal{L}$ *is a set of labels and* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{L}$ *is a triple of the form* $\langle v1, v2, l \rangle$ *(v1, v2* $\in V$ *and* $l \in \mathcal{L}$*) is called* directed labeled graph.

Definition 17 (Dependency graph) *Let* $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *be a directed labeled graph, then G is a* dependency graph *for the program* Π*, iff* $\mathcal{V}$*=HB(*Π*) and* $\mathcal{L}$ *= {+, -} and for all* $a, b \in V$ *the condition* $\langle a, b, + \rangle \in \mathcal{E}$ *holds, whenever a rule* $r \in \Pi$ *with* $a \in head(r)$ *and* $b \in body^+(r)$ *exists, or* $\langle a, b, - \rangle \in \mathcal{E}$ *holds, whenever a rule* $r \in \Pi$ *with* $a \in head(r)$ *and* $b \in body^-(r)$ *exists. A directed graph* $G^+ = \langle V, E^+, L \rangle$ *where* $E^+ = \{\langle v1, v2 \rangle | \langle v1, v2, + \rangle \in E\}$ *is called* positive dependency graph *w.r.t. G. A directed graph* $G^- = \langle V, E^-, L \rangle$ *where* $E^- = \{\langle v1, v2 \rangle | \langle v1, v2, - \rangle \in E\}$ *is called* negative dependency graph *w.r.t. G.*

Definition 18 (Typed Dependency Graph) *Let* $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *be a dependency graph for the program* Π *with the alphabet A=(P, V, C), then G is a* typed dependency graph *for* Π*, if every* $v \in \mathcal{V}$ *has the type of an element of V—*$type(v) \in V$*. Type(v) represents a function for identifying to which element of A a node is referring to.*

The basis for the considered graphs in this thesis is provided by typed dependency graphs (cf. Definition 18). For each directed graph (DG), i.e., a typed dependency graph for the program Π, a set of paths can be constructed involving a set of vertices (nodes) and edges.

Definition 19 (Path) *A path p in a directed labeled graph* $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *is a sequence p=*v_1*, ...,* v_n *(*$n \geq 1$*) where for each element* $v_i \in p$ *and* $v_{i+1} \in p$*—with* v_i*,* $v_{i+1} \in V$ *and* $1 \leq i < n$*—an edge* $e_i = \langle v_i, v_{i+1} \rangle \in \mathcal{E}$ *exists.*

In some cases the restriction to acyclic graphs is useful, because it reduces the risk of infinite loops or hardly visualizable constellations.

Definition 20 (Acyclicty) *A directed graph G is called acyclic if no path p with p=*v_1*, ...,* v_i*, ...,*v_j*, ...,* v_n *exists in G where* $v_i = v_j$ *for* $i \neq j$*.*

Graphs for Deductive Programs

The approaches presented in the following sections are based on typed dependency graphs $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ (cf. Section 5.2). All elements of Π are visualized as graphical nodes of *V* and interconnected by edges of *E*. In this section, some specific definitions are introduced being used by all following graphical approaches.

Definition 21 (Elementary nodes) *Let A=(P, V, C) be the alphabet of the program* Π *and* $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *be a graph of* Π*, then each visualized element has a correspondence to an elementary node* $v \in \mathcal{V}$*.*

Definition 22 (Standard gateway node) *Let A=(P, V, C) be the alphabet of the program* Π *and* G_Π *be a graph representing* Π*, then node g is named a* standard gateway node*, iff it represents an element of A being attached to the tuple* $l_i = \langle V_i, P_i, C_i \rangle$*—*$V_i \subseteq V$*,* $P_i \in P$*, and* $C_i \subseteq C$*—referring to a literal of rule* r *(where r is a rule of* Π*) in* G_Π*.*

Definition 23 (Rule gateway node) *Let A=(P, V, C) be the alphabet of the program* Π *and* G_Π *be a graph representing* Π*, then node g is a* rule gateway node*, iff it represents an element of* $r \in R$ *where R are the rules of* Π*.*

Definition 24 (Gateway node) *Let* $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *be a graph representing the program* Π*, then each node* $g \in \mathcal{V}$ *represents a* gateway node*, iff g represents a* standard gateway node *or rule gateway node of* Π*.*

Elementary nodes are used for visually representing the elements of the alphabet of Π, whereas gateway nodes represent placeholders for being the elements of ASP rules. No redundancy of elementary nodes is allowed in the graph G of Π whereas gateway nodes represent different applications of the same alphabet element, e.g., applying different term assignments or negations for particular rules.

A requirement for all nodes is the retention of the used naming structures provided by the program Π to be visualized. A literal has to be named as similar as possible in the graphical representation (node) as in the code. This is very important for the understanding of the semantics of Π.

Definition 25 (Edge) *An edge of the graph G is a tuple of the form* $e = \langle S, T \rangle$ *where e points from a source node S to a target node T, and S, T are gateway nodes of G.*

In ASP programs, relationships or dependencies between rules are only implicitly existent. In the graphical visualization, edges (cf. Figure 5.5) can be used to make them syntactically explicit to users. Each edge pointing to a node declares a dependency of this node to the origin of the edge. Detailed definitions of dependencies are introduced in Section 5.3.

5.3 Propositional Answer Set Programs

In the following sections, the previously presented ER-diagram based approach—cf. Section 4—is revised for supporting the visualization of deductive answer set programs. In particular, the visualization is adapted to allow the representation of arbitrary rules (cf. Section 5.4 and Section 5.5). In this section, only *propositional logic ASP* programs are considered by step-wise translating ASP programs Π to a graphical representation. This restriction of the ASP language allows the concentration on basic program structures and reduces the necessity for an own instance-based visualization layer (or even model). An example for propositional answer set programs is shown in Equation 5.2.

$$\Pi_{5.2} = \begin{cases} r1 : summer \vee winter & \leftarrow . \\ r2 : warm & \leftarrow oven_on. \\ r3 : warm & \leftarrow summer. \\ r4 : \neg icy & \leftarrow warm. \\ r5 : & \leftarrow \neg icy, winter, oven_on. \end{cases} \tag{5.2}$$

Definition 26 (Propositional Answer Set Programs) *Let A be the alphabet of* Π *the form A=(P, V, C), then* Π *is a propositional logic program, iff for each predicate p occurring in* Π *it holds that arity(p)=0.*

Representations of Language Concepts

In the following, we shortly introduce several graphical language concepts that are necessary to visualize answer set programs.

Nodes & Edges for the Representation of Rules

Nodes. Each rule of the program Π consists of a set of body and head literals. Body and head literals are visualized as gateway nodes. These nodes can be *positive* (unnegated) or *negated.* Each positive node is represented as node with dotted border without any additional labeling. Negations are expressed by labels in boxes. The (1) classical negation (¬) and the (2) default-negation (*not*) are supported by our approach. Negated nodes are visualized like positive literals, but require an additional label ¬ (negation) or *not* (default-negation) as demonstrated in Figure 5.2. The label for default-negations can only be used for body literals.

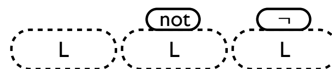


Figure 5.2: Nodes representing positive, default-negated, and strong negated literals

Edges. For gateway nodes a set of directed relationships forming dependency structures are expressed by edges. Each node targeted by an edge is dependent on the source of this edge. This dependency relationship has some implications on the containment of literals (visualized as nodes) in answer sets, which are thereinafter introduced. Dependency modalities result from the used edge variant and their associated semantics. These dependency variants and concrete edge types are introduced in the following definitions.

Definition 27 (Dependency) *Let e be an edge of the form $e = \langle S, T\rangle$ in the graph G_Π representing the program Π, then S (the* source*) is* directly dependent *on T (the* target*), iff T can be applied via S and the connecting edge e—S is a predecessor of T in the graph. T is* indirectly dependent *on a set of gateway nodes N, iff T can be applied by any edge $\langle N_i, T\rangle$ where $N_i \in N$. In the graph, T is a successor of one or more gateway nodes in N.*

In the simulation of answer sets a *direct dependency* resulting from an edge $e = \langle S, T\rangle$ implies that for every answer set X_i of Π the following two conditions hold $S \in X_i \Rightarrow T \in X_i$ holds for every unnegated or strong negated *S*. Direct dependencies can be used to form a set of dependencies—i.e., a rule being dependent on all of its body literals—forms a conjunction of dependencies. In the simulation of answer sets, indirect dependencies in contrast imply for every answer set X_i of Π that for the set of gateway nodes $N \in \Pi$ the condition $N \in X_i \Rightarrow T \in X_i$ holds as well as any N_s with $N_s \subset N$ exists satisfying $N_s \in X_i \Rightarrow T \in X_i$. More details on dependency concepts are introduced in forthcoming sections.

Two different categories of edges are necessary to express relationships forming direct and indirect dependencies in Π: E_D, and E_{SR} edges.

Each body literal or head literal—based on Definition 6—represented as graphical nodes connected with the rule comprising this literal by making use of E_D edges (this is a direct relationship).

Definition 28 (E_D edge) *Let $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L}\rangle$ be the graph for a program Π, then an E_D edge is a directed edge of the form $e = \langle S, T\rangle$ where T is directly dependent on S, and S=g and T=r (named a* consummation *of g by r) or S=r and T=g (named a production of g by r) for the rule $r \in R$ (where R are the rules of Π) and the standard gateway node $g \in V$.*

According to Definition 28 two variants of E_D according the node type of the edge source and target. (a) The E_{D-Body} edge is used for connecting body literals (denoted as gateway nodes) of a rule *r* in Π. As a consequence, *r* is directly dependent on the information content of the connected body literal. If a set of E_{D-Body} edges for *r* is used, each literal as source of these edges represents an $\wedge$-connected body literal of *r* in Π, e.g., $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ where L_i with $0 < i \leq n$ are body literals of *r*. (b) The E_{D-Head} edge connects head literals of *r* with *r*. The gateway nodes targeted by these edges therefore are implicitly $\vee$-connected head literals directly depending on *r*. In Figure 5.3 an example is given highlighting these implicit conjunctions and disjunction of nodes. The gateway nodes representing the body literals *hot* and *summer* are implicitly $\wedge$-connected. The gateway nodes representing the head literals *swimming* and *sleeping* are $\vee$-connected.

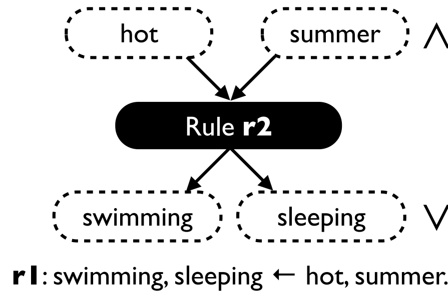


Figure 5.3: An example of implicit connectives for gateway nodes

Definition 29 (Candidate answer set) *Let* Π *be a program with the rules R with* $R_C \subseteq R$ *(*R_C *is the set of constraint rules of* Π*), then a model of* Π *according Definition 10, which has not been validated against* R_C*, is called a* candidate answer set *of* Π.

Definition 30 (E_{SR} **edge)** *A directed edge e of the form* $e = \langle S, T\rangle$ *represents an* E_{SR} *edge, iff S=g1 and T=g2 where g1 is a standard gateway node representing head literal* l_1 *and g2 is a standard gateway node representing body literal* l_2*. T is indirectly dependent on all nodes* S_i*—arbitrary gateway nodes—for which an edge of the form* $\langle S_i, T\rangle$ *exists.*

Definition 31 (E_{SR} **satisfaction)** *Let e be an* E_{SR} *edge of the form* $e = \langle S, T\rangle$*, then the indirect dependency of T on S causes that the condition* $S \in AS \rightarrow T \in AS$ *(AS represents a candidate answer set of a program* Π*) has to hold—this is abbreviated by T is satisfied by e. The relationship between S and T is named* sender-receiver *relationship.*

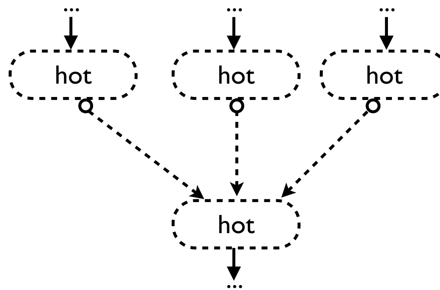


Figure 5.4: Three E_{SR} edges pointing from head literals to a body literal of another rule

In Figure 5.4 an example is given using E_{SR} edges as specified in Definition 30 and 31. Three body literals *hot* are connected via E_{SR} edges with a body literal *hot* of another rule. The indirect dependencies of E_{SR} edges imply that whenever one of the head literals of the first rule can be reached (is contained in the inspected answer set), the body literal of the successor rule can be reached. The E_D edges in contrast require that every connected predecessor node can be reached—is element of the respective answer set.

As body literals could be indirectly dependent on head literals of all applicable rules of Π, a differentiation between body literals as edge targets (*{*$b_1, \ldots, b_n$*, not* $b_{n+1}, \ldots,$ *not* b_m*}* in Definition 6) and available sources (*{*$h_1, \ldots h_k$*}* of Definition 6) is necessary to recognize possible rule applications in the visualization. The textual representation does not explicitly, i.e., visually, link these source nodes with a target node. Contrary, in our approach a set of sender-receiver relationships are used for explicitly visualizing such constellations (cf. Definition 30) with the help of E_{SR} edges. Graphically it has to be differentiated between default-negated and all other head literals targeted by E_{SR} edges. All body literals without default-negation have to be targeted by at least one E_{SR} edge pointing to them in order to allow the application of the rule. For this case, the special edge variant $E_{SR-Add.}$ is used. Contrary, each edge targeting a default-negated body literal (as gateway node) can inhibit the execution of the rule comprising this body literal. As consequence, each candidate answer set or subset of it comprising an instance being element of a literal tuple corresponding to a default-negated body literal of a rule, is excluded from being applicable for the rule execution.

	Dependency	**Origins**	**Semantics**	**Source**	**Target**
E_{D-Body}	Direct	All	$\wedge$	Std. gateway	Rule gateway
E_{D-Head}	Direct	All	$\vee$ (only for disj. programs)	Rule gateway	Std. gateway
$E_{SR-Add.}$	Indirect (Bound to a literal)	Any	Sharing resource (additively)	Std. gateway	Std. gateway
$E_{SR-Subtr.}$	Indirect (Bound to a literal)	Any	Sharing resource (subtractively)	Std. gateway	Std. gateway

Table 5.1: Summary of variants of edges based on [4]

For providing a better overview and syntactic differentiation based on [4], the presented edges are summarized in Table 5.1 and visualized in Figure 5.5. The following five characteristics are inspected: On the one hand (i) the dependency relationship between the source and the target of an edge (cf. Definition 27), (ii) the consummation *Origins* of nodes (stating, if a target node has to consume from *all* source nodes from connected edges, or can consume from *any* applicable source), and (iii) general semantical differences, i.e., implicit connectives or the general functionality of an edge, are used to compare the discussed types of edges. On the other hand (iv) the type of nodes which can act as *Source* of a directed edge variant, and (v) the type of nodes which can act as *Target* of a directed edge variant are inspected. The characteristics (i) to (iii) are related to semantical properties whereas (iv) and (v) highlight syntactical differences.

According these five characteristics, E_D edges can be classified as edges providing direct relationships between a source and a target. Every specified source for a node connected via E_D edges, has to be reachable to involve the target—denoted by the number of alternative *origins* for the satisfaction of a dependency, which is exactly one for each E_D edge. The set of standard gateway nodes from which an E_{D-Body} points to a rule gateway node, is implicitly $\wedge$ connected.

Contrary, standard gateway nodes being targeted by E_{D-Head} edges, are $\vee$-connected—this is only possible for disjunctive programs. Especially of interest are the $E_{SR-Add.}$ and $E_{SR-Subtr.}$ (cf. Table 5.1 and Figure 5.5) edges providing an indirect dependency relationship causes the possibility of multiple origins (from any origin a consummation is possible) as predecessor nodes. It is undefined from which particular origin the body literal is actually consumed. If not all body literals of a rule are satisfied sufficiently by at least one providing $E_{SR-Add.}$ targeting them or an $E_{SR-Subtr.}$ edge points to a default-negated body literal, no consumption is undertaken—the rule is not applied.

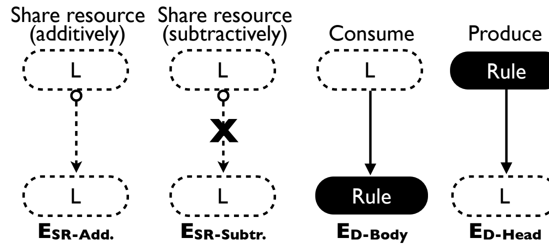


Figure 5.5: Types of edges

All four discussed edge variants are exemplary shown in Figure 5.5. On the left the edges for sender-receiver edges are visualized in additive and subtractive variants. In particular, these edges share the resource L with another rule. On the right side edges for directed dependencies are used for body literals and head literals of a rule.

Putting it all together

A rule itself represents a rule gateway node. Rules are essential to be visualized as they are essential for understanding the underlying code of Π. Rules represent the only structuring facility within Π, as the ASP language does not comprise structuring concepts such as objects or functions. However, such encapsulations (could be separation of concerns) are valuable for debugging Π and for dynamically applying program parts.

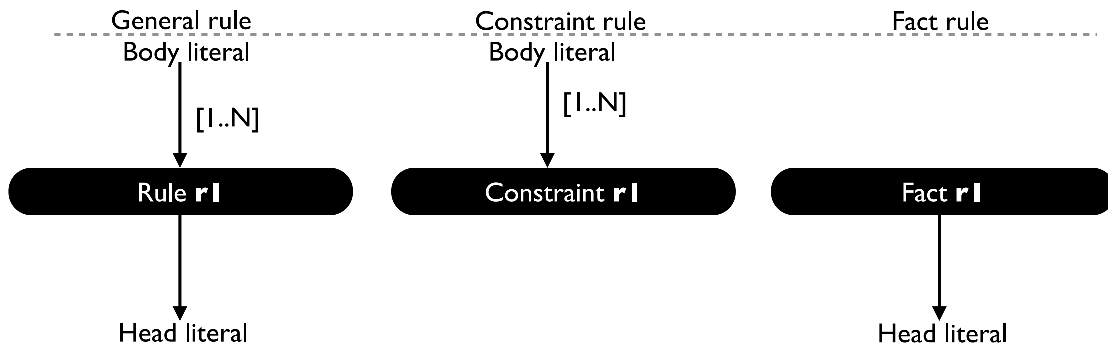


Figure 5.6: Different variants of rules

Special forms of rules are facts and constraints. Facts are entry points for the graphical representation, as they typically do not depend on other rules. Consequently, in this approach facts are integrated in an overall graphical visualization in lieu of focusing on the modeling of their schema (cf. Section 4). In particular, facts do not require any incoming edge E and produce outputs that can be used by several other rules. Constraint rules do not contribute any literals to a candidate answer set. Such rules restrict the number of possible answer sets by eliminating inconsistent or unintentional candidate answer sets. As constraint rules have to be applied at any place of the graph, their graphical visualization is definitely more challenging than the visualization of facts or other rules. All three types of rules are shown in Figure 5.6. The different types may be distinguished by their labeling and their ability of handling incoming and outgoing edges.

Summary

There exist elementary nodes and gateway nodes. Elementary nodes are used for explicitly visualizing the alphabet of a program Π. Gateway nodes are used for representing alphabet elements being used within a rule—predicates, variables and constants, and the rule itself. Gateway nodes—except rule gateway nodes—may be decorated with additional properties such as negations.

Moreover, several different edges are necessary to graphically represent an answer set program. An overview of the defined edges according the following characteristics are given in Table 5.1. In particular, the differentiation of direct relationships and sender-receiver relationships is necessary. Direct relationships result from explicit dependencies, i.e., a rule comprising a set of literals. Sender-receiver relationships are implicit relationships resulting from all possible rule applications.

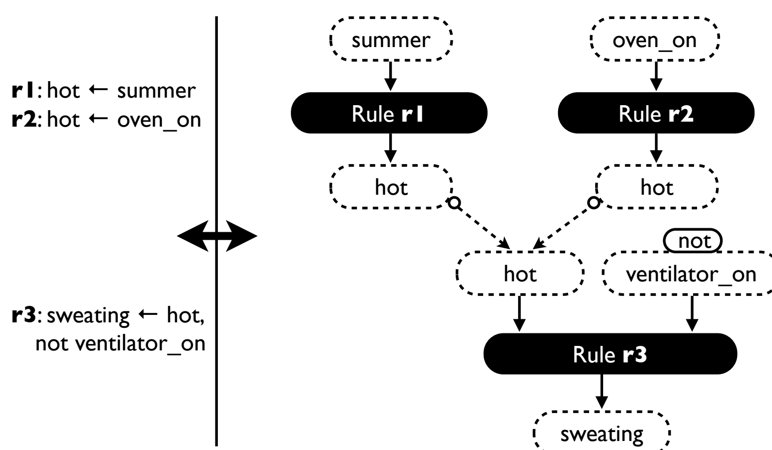


Figure 5.7: A first visualization of a deductive ASP program

An example making use of these nodes and edges is given in Figure 5.7. The program consists of three rules where the rules *r1* and *r2* exactly contain one body and one head literal.

The rule *r3* contains a second body literal which is default-negates (*not ventilator_on*). Each literal represented by a gateway node (cf. right side of Figure 5.7) is connected with a rule by using an E_D edge, e.g., the body literal *oven_on* as gateway node is connected to rule *r2*. As rule *r3* can consume *hot* from rule the body literal *hot* of rule *r1* and *r2* (two E_{SR} edges are pointing to it), *r3* is therefore applicable whenever rule *r1* and *r2* are applicable and no rule holding a head literal *ventilator_on* is applicable. As a rule is only applied when all default-negated literals are not targeted by any $E_{SR-Add.}$ edge (cf. *r3* in Figure 5.7) and all others are targeted by at least one $E_{SR-Add.}$ edge, *sweating* is indirectly dependent on *r1* and *r2*. Furthermore, it has to be noted that the nodes *hot* may be considered as redundant. The reduction of redundancies is discussed in the Section 8.

Program Arrangement

To be able to construct a program Π consisting of a set of rules, the head literals of each rule—the set of literals $[C_I, C_N]$ (N is one for non-disjunctive programs)—are connected with body literals of other rules. The dependencies of rules were defined in the previously discussed concepts. The usage of different rule types and their interactions is introduced in this section.

To be able to meaningfully construct a program as a visual model, it is essential to identify which nodes can act as *root nodes*. This approach can also be reused for undertaking reverse engineering of existing ASP codes to such graphical models. Root nodes represent a starting point in the visual simulation of ASP programs and represent nodes in the graph which are not dependent on other nodes. Fact rules or standard rules which only hold default-negated body literals can be represent as root nodes (gateway nodes).

The graphical representation is intended to be read from top to bottom—loops or recursions can influence this reading guideline. As a consequence, root nodes should be placed at the top of the graph. This adoption is used for improving the ease of recognition of essential elements. Typical root nodes are facts which are the basis for applying other rules. Not only facts can act as root nodes, but also other rules which only hold default-negated body literals. This implies that any rule with one or more positive literals (atoms) cannot be used as root node as these atoms cannot be satisfied without the assistance of another rule.

A path—as introduced in Section 5.2—expresses a particular sequence, such as a rule sequence. With each move along edges (without recursion this is typically downwards) on a path from root nodes to final nodes, nodes without any outgoing edge, of the graph *G*, a chain of dependencies is constructed. A final node is dependent on all other nodes on the path. In Figure 5.8, all following nodes are dependent on the root nodes $L1$ and $L2$. The final node $L5$, therefore, is dependent on all previous nodes ($L1, L2, L4$) except $L3$ which is not reachable over any path from $L5$. Although nodes might be involved in several paths, these paths are not directly influenced by each other. The only influence of other paths occurs when $SR_{Subtr.}$ edges point to a node of another path. Then the forward processing of a path might be interrupted at this node. As a consequence, this visualization implicitly constructs a reduct of Π from which duplicates have to be removed to construct valid answer sets.

Moreover, the path construction in the graph is comparable to the logical implication—$r_{level-1} \rightarrow r_{level-2} \rightarrow \ldots \rightarrow r_{level-n}$. The satisfaction of preconditions by predecessor rules automatically leads to an application of this rule (the successor rule) as well. The application of

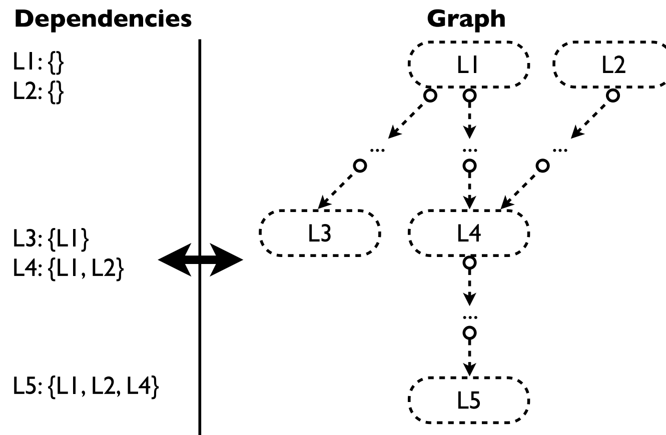


Figure 5.8: Dependencies of nodes

a successor rule could, but needs not, automatically prove that the predecessor rules are satisfied as well. This dependency behaviour can be simplified to the following definition: Each rule (representing the vertex $v \in G$) is dependent on each of its direct predecessors on all $p_i \in G$ which satisfy $v \in p_i$—direct predecessors have the distance of one edge.

Each constructible path p of a complete graph G_Π conforming to a valid answer set program Π, expresses an ASP model of Π (cf. Definition 10). By eliminating existing duplicates from it and removing ASP models violating constraints, a reduct Π^p results—an answer set of Π. Invalid paths p comprising misspecified edges (e.g., incorrect sources or targets are used), misspecified nodes, misspecified inferences—not corresponding to Π—or incomplete node and edges sequences (missing elements) cannot be transferred to valid answer sets.

Contrary to facts (and other root nodes), constraint rules do not provide literals as conclusions (head), but only body literals. Therefore, constraint rules cannot be placed as root nodes, but rather as final nodes. Although the constraint rules have to be fulfilled at any point of the computation of Π, the final position of constraint rules is sufficient as they are enforced to be applied. This can be only achieved at the bottom of a graph where the state is inherited from predecessor rules. All other rules are positioned between root nodes (e.g., facts) and final nodes (constraint rules or final nodes when no constraint rule is available) in a desired execution order. The interrelations are achieved by using E_{SR} edges.

These three blocks of rules visualized as graphical nodes are the *layers of* Π. These layers are intended to be used for all programs. In Figure 5.9 these three layers are graphically visualized. Facts *r1*, *r2*, and *r3* serve as root nodes (like facts, cf. Section 5.3), from which *standard rules* (*r4*, *r5*, ...) are applied, and finally ending with constraint rule *r6* (cf. Section 5.3) on the third layer. If there are no constraint rules, the last layer remains empty.

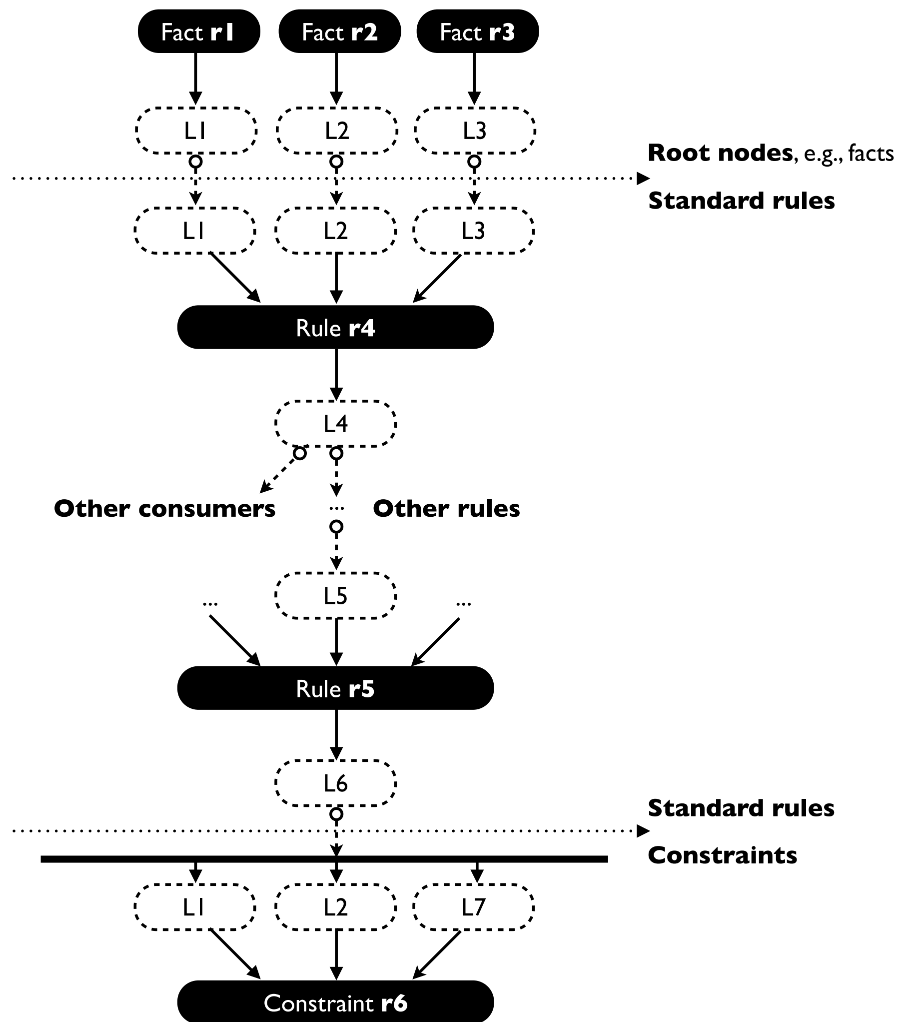


Figure 5.9: The layers of an ASP program graph

5.4 Programs of Literals with Maximum Arity One

Unary logic predicates as literals of ASP rules have to be visualized differently than propositional programs. This strongly influences the complete visualization of ASP programs as well. Therefore, the proposed concept for deductive propositional answer set programs is extended by the introduction of arguments (variables, constants) and their equalities or inequalities. The arity is limited to one as the visualization complexity strongly increases by introducing concepts for higher arities. It is, therefore, the aim of this section to provide a basis for the usage of predicates with higher arities than zero. In the second part of this section the visualization is enhanced as the newly introduced features require some visual adoption.

Variables & Constants

The basic visualization approach of unary literals needs concepts to show constants (instances). This is necessary as each used variable may correspond to a set of constant values in the execution or simulation of an ASP program. Constant values are of the form *L(a)* where *a* is the constant and L is a predicate. Recall, that variable names are written with upper case and constants with a lower case starting letter.

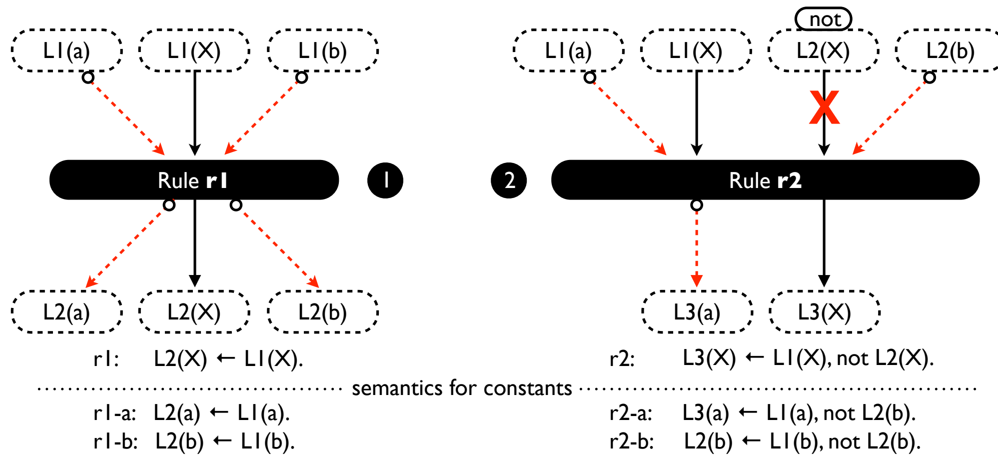


Figure 5.10: Two unary logic answer set program examples—one applicable (1) and one inapplicable (2)

In the example (1) of Figure 5.10 the rule *r1* makes use of the literal *L1(X)*. For each constant value corresponding to *X* that satisfies *L1(X)*, *L2(X)* is the consequence. This implicitly requires a parsing or identification of "constant to variable relationships" in respect to previously inferred literals. Therefore, for a rule *L2(X) ← L1(X)* constant-specific rule applications are existing (e.g., *L2(a) ← L1(a).*) For the concrete examples of constant *L1(a)* and *L1(b)*—provided by predecessor rules—the literals $L2(a)$ and $L2(b)$ are returned. For both of these constants the rule can be applied independently. Whenever a constant satisfies all body literals holding a specific variable (e.g., X) as argument, it corresponds to this variable in this rule—a literal gateway for this constant is added to all usages of this variable (e.g.,$L1(a)$). If there exist unbound variables in the head (variables in body literals which do not correspond to a variable in any literal of the head—or vice versa) of standard rules, only some constant-specific literal gateways are defined which do not correspond to any constant. Rules making use of default-negations require a different handling of constants in the visual representation. In particular, the constants violating such body literals avoid the rule application. Furthermore, an additional edge variant (E_C)—the dashed red edges—that allows a differentiation of applicable constant values and general dependencies, is necessary. In (2) of Figure 5.10 *L2(b)* violates *not L2(X)* as no evidence of any constant value matching *L2(X)* is allowed. This violation could only be avoided by defining that b may not match X. This matching is achieved by only attaching applicable constants to literals of a rule. It is necessary to note that for fact rules a variable is not

be available. Therefore, only instance-specific literals are produced.

Definition 32 (Equality) *The binary predicate* equal(X, Y) *implies that* $C_X \subseteq C_Y$ *and* $C_Y \subseteq C_X$ *holds where X and Y are variables,* C_X *is a set of constants matching X, and* C_Y *is a set of constants matching Y. This* equality *predicate for X and Y is denoted by* $X = Y$.

Definition 33 (Inequality) *The binary predicate* unequal(X, Y) *implies that* $C_X \nsubseteq C_Y$ *and* $C_Y \nsubseteq C_X$ *where X and Y are variables,* C_X *is a set of constants matching X, and* C_Y *is a set of constants matching Y. This* inequality *predicate for X and Y is denoted by* $X \neq Y$.

Equalities and inequalities of variables may be defined to avoid the application for unintended constellations. Such inequality and equality expressions represent binary predicates with a simplified notation, e.g., $X = Y$ and $X \neq Y$ could be used instead. However, a solver-specific understanding and handling of such expressions is necessary.

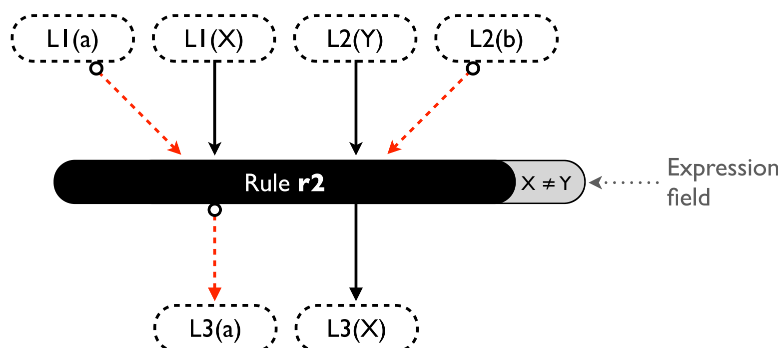


Figure 5.11: The usage of an expression field

Equalities and inequalities require additional language concepts. It has to be noted that an equality or inequality is valid within a complete rule application. This is notable as the definition of such equalities or inequalities are placed in the rule body in ASP. Typically such equalities or inequalities are used in ASP to ensure that two variables are not matched with the same constants—e.g., the inequality of $X \neq Y$ states that the literals *L1(X)* and *L2(Y)* have to be addressed by different constants. The rules in ASP are structurally independent of each other—only values are passed to other rules or are consumed from other rules. The behaviour of the rule application is, therefore, highly influenced by such definitions. As a consequence, the equalities and inequalities of variables and all other constraint expressions necessary within a rule, are directly integrated in the graphical rule definition. For this purpose, an *expression field* (cf. Figure 5.11) is added to the rule's node. Such expressions could also hold other expressions or a set of expressions. Complementary expressions could lead to a non applicability of rules. As the number of such expressions in Π is typically small, the placement in the rule node provides sufficiently space for their visualization. In Figure 5.11 the rule *r2* holds two body literals—*L1(X)* and *L2(X)*. Both have to be satisfied to produce *L3(X)*. Additionally, X and Y have to be unequal ($X \neq Y$). This means that at least one instance (a constant) being attached to the literal *L2* has to exist which does not not correspond to X (used as argument for *L1*) to satisfy the rule.

Another interesting difference to propositional ASP can be seen in the usage and definition of facts. In propositional ASP programs no differentiation can be made between general statements and concrete instances. For example in propositional logic programs the predicate *warm* states that it is generally warm, whereas in first-order logic ASP programs *warm*($room_a$) can more specifically state that it is only warm in $room_a$ (cf. $\Pi_{5.3}$ in Figure 5.3). Graphically, a set of constants conforming to the same argument of a literal within a fact, can be aggregated to a single visual rule representation (an abstract fact definition). The specific variables are attached to the arguments of a literal (concrete facts).

$$\Pi_{5.3} = \begin{cases} r1 = & warm(room_a) \leftarrow . \\ r2 = & warm(room_b) \leftarrow . \\ r3 = & warm(room_c) \leftarrow . \end{cases} \tag{5.3}$$

In contrast to propositional answer set programs, for unary logic programs another differentiation is necessary. In unary logic programs each usage of a literal *L* is customized by the used variable usage *Vu* for a rule *r*. Therefore, different usages of the same literal expression, e.g., *Fly(X)* from the expression *Fly* can refer to a different application behaviour in a rule context. Therefore, in this approach for unary logic programs the following concepts have to separated from each other: Variables and their usages (variable usages); Literals and their application for a rule (literal gateway); Constants and their application for a variable usage in a rule.

A variable usage is an application of a variable in a particular rule of a program Π, a literal gateway is an application of a literal for a particular variable usage, and a constant gateway is an application of a constant for a particular variable usage.

Definition 34 (Variable usage) *Let* $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *($\mathcal{V}$ is a set of vertices, $\mathcal{E}$ is a set of edges, and $\mathcal{L}$ is a set of labels) be a graph for the program* $\Pi = \langle P, V, C \rangle$ *(P is a set of predicates, V is a set of variables, and C is a set of constants). Then a variable usage of the rule* $r \in R$ *represents a vertex* $vu \in \mathcal{V}$ *forming the tuple* $\langle v, r \rangle$ *where the variable* $v \in V$ *and the rule* $r \in \Pi$.

Definition 35 (Literal gateway) *Let* $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *($\mathcal{V}$ is a set of vertices, $\mathcal{E}$ is a set of edges, and $\mathcal{L}$ is a set of labels) be a graph for the program* $\Pi = \langle P, V, C \rangle$ *(P is a set of predicates, V is a set of variables, and C is a set of constants). Then a literal gateway* $lg \in \mathcal{V}$ *representing a literal* $l \in r$ *where the rule* $r \in \Pi$*—forming a tuple* $\langle l, r \rangle$.

Definition 36 (Constant gateway) *Let* $G_\Pi = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *($\mathcal{V}$ is a set of vertices, $\mathcal{E}$ is a set of edges, and $\mathcal{L}$ is a set of labels) be a graph for the program* $\Pi = \langle P, V, C \rangle$ *(P is a set of predicates, V is a set of variables, and C is a set of constants). Then a constant gateway of the rule* $r \in \Pi$ *represents a vertex* $cg \in \mathcal{V}$ *forming the tuple* $\langle c, vu \rangle$ *where the constant* $c \in C$ *and the variable usage* $vu \in \Pi$.

The assignment of elements to tuples representing variable usages, literal gateways, and constant gateways are achieved by using typed references and edges of the graph *G* representing a particular answer set program Π.

Optimization for Program Analysis

In the previous, section relationships and instant-specific definitions have been integrated in the same visualization. This is an asset for understanding the behaviour principles of a program, but it may be difficult to recognize variable usages over a range of different variables and constants. This may be explained by the need to link the root node providing constants with consuming nodes (a rule consuming this literal). So, if a certain constant is used by one of the first nodes from the top and used by one of the last nodes from the top, at least one edge crosses the overall visualization canvas. The longer the used edges, the more often edges will cross each other and the higher the visualization complexity will be. Therefore, it can be advisable to separate these levels from each other. This is achieved by using a meta level model and several instance level diagrams. The instance level diagrams are specific instance based reductions of the program Π whereas the the visualizations on the meta level do not visualize any constants. So, the meta-level model is responsible for designing Π. The proposed enhancements in this section are intended to improve the applicability for analysis. This is achieved by a clear coupling of information, the focus on essential nodes and a clear reasoning visualization why a certain rule is applied or a certain constraint rule is violated.

Meta Level

The visualization (model) on the *meta level* provides a complete overview of the application design. It visualizes all rules using variables. All constants addressing these variables are excluded from the visualization. This view on an ASP program is intended to be used for developing paths for the application of rules. It is called the "meta level" as it provides a view on a higher abstraction level than instance-based models (or models including instances).

The program Π_{fl1} of Figure 5.12 introduces two root rules: *r1* and *r3*. If a product *X* does not satisfy *cheap(X)*, it satisfies *expensive(X)*. If *X* does not satisfy *expensive(X)*, it satisfies *cheap(X)*. Therefore, a product cannot be *cheap* and *expensive* at the same time. However, there can be different answer sets that define this differently for the same product. If a product is *cheap*, the customer buys it. However, if the customer realizes that the product has already been used (*not used* not satisfied), we do not know how he decides. The customer does not buy ($\neg buy(X)$) an expensive (*expensive*) product. Furthermore, the buying of any product is legally prohibited and, therefore, disallows answer sets with this literal constellation. More general constraint rules can be used as well, that disallow the buying of products (*buy(X)*) when the item is sold out (*sold_out(X)*)—cf. rule *r5* in Figure 5.12.

Instance Level

The *instance level* visualization opens the possibility of reducing the overall visualization complexity in two ways: First, constants that are not part of an answer set for a particular scenario and are not necessary for a constraint rule visualization, are left out. Second, structures (nodes, edges) which are not involved in producing an answer set and which are not necessary for understanding the violation of a constraint rule, need not be visualized. This view, therefore, allows the reasoning why a certain answer result could or could not be returned without involving details that are not necessary for understanding the behaviour of Π for this particular case.

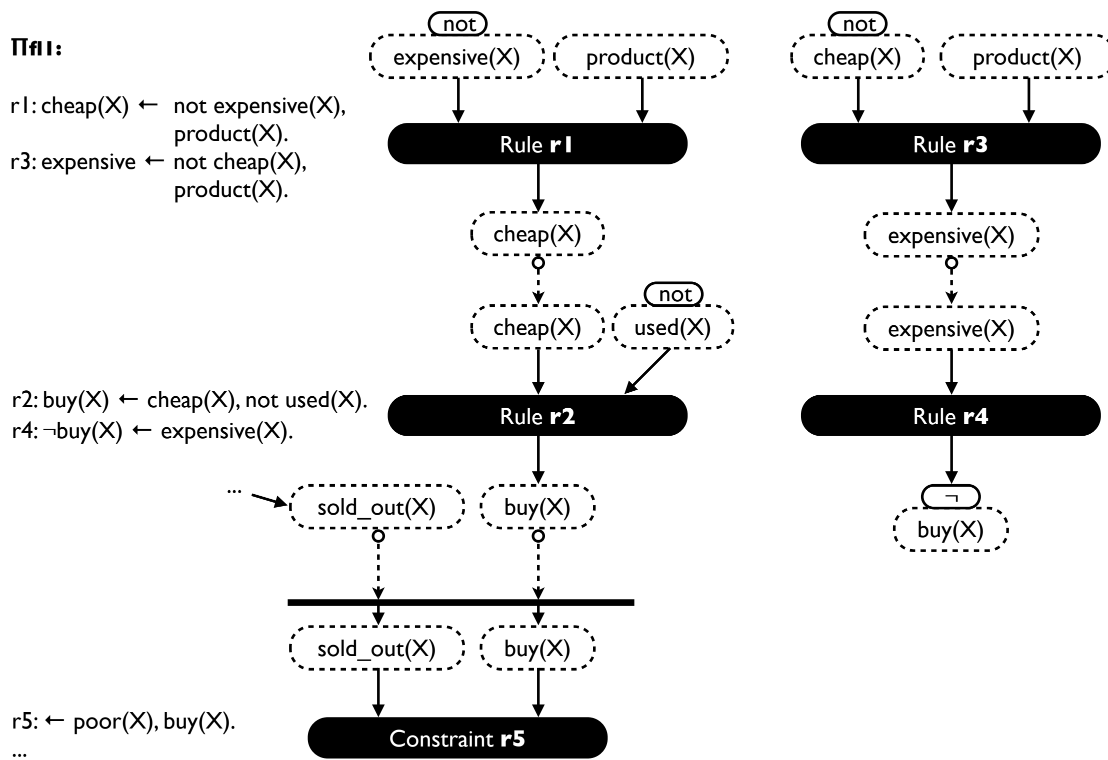


Figure 5.12: A model on meta level representing the ASP program Π_{fl1}

To reduce the visualization complexity, it is necessary to aggregate similar rules, e.g., two rules with the literals *product(a)* and *product(b)* can be abstracted to *product(X)*. This is achieved by visually attaching constants to variables, e.g., *a* and *b* are attached to *X* in a rule. For this purpose, boxes for instances are added to literal gateway nodes representing body or head literals.

In Figure 5.13 the two answer sets *ASI* and *ASII* for program Π_{fl1} of Figure 5.12 are shown. Several facts were not visualized in the meta level model. The following three facts have been used: *product(a)*, *product(b)*, *used(a)*. These facts allow the processing of Π at instance level as the variables cannot provide particular answer sets. The duplicate literal gateways *expensive(X)* between rules *r3* and *r3* are redundant. However, by adding new rules, edges to the literal representing a body literal of *r5* may be necessary. As a consequence more constants may be added to this node as well. Therefore, this redundancy is accepted at this point.

In the visualization Π_{ASI} of Figure 5.13, the rules *r3* and *r4* of program Π_{fl1} do not have an effect on the answer sets. Therefore, they are not visualized. As *a* and *b* are both products and there is no knowledge of *a* and *b* being expensive, both are applicable for rule *r1*. This rule returns that both are *cheap*. This makes them eligible for being used by rule *r2*. However, this rule has another body literal (*not used*) which is only satisfied by *b*. The knowledge of *used(a)* disallows its usage for *r2*—marked with a red cross. This visualization is essential as it provides the reasoning for certain results. In following steps these invalid values need not be handled any more. As the result of r3 (*buy(b)*) does not violate the constraint rule of *r5*, the answer

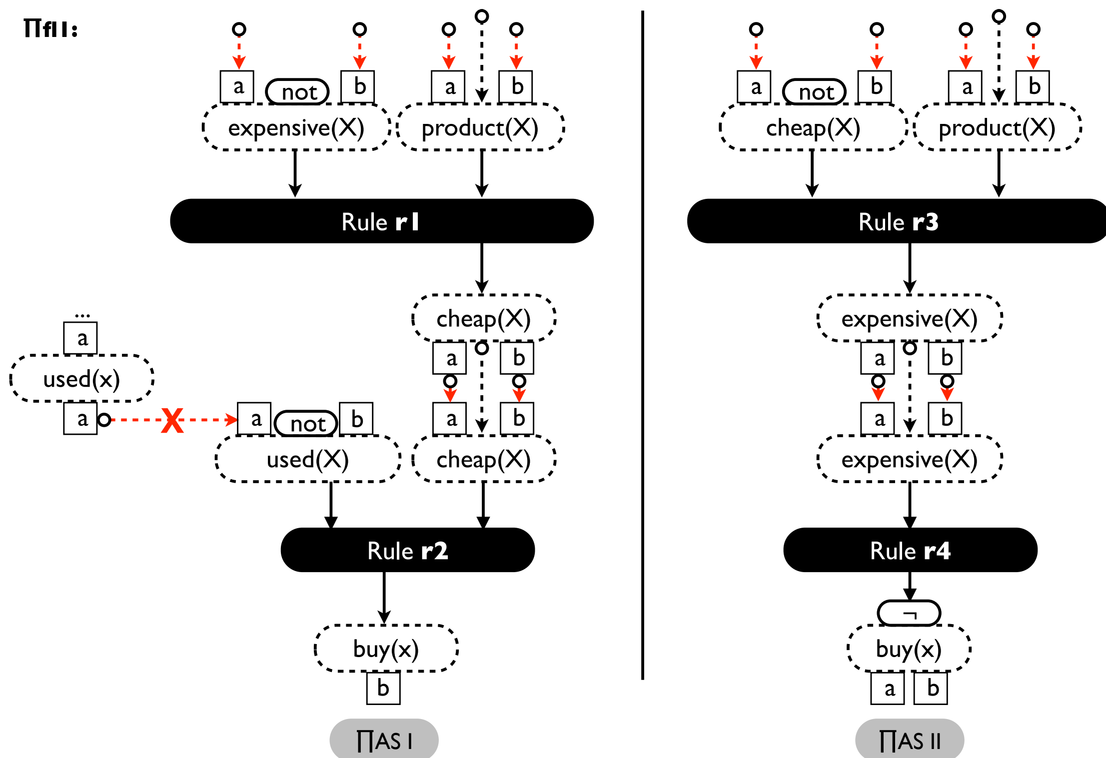


Figure 5.13: An instance level model representing the ASP Π_{fl1} with the constants a and b

set is valid. The result of *ASI*, therefore, is: *ASI= {product(a), product(b), used(a), cheap(a), cheap(b), buy(b)}*.

In the visualization Π_{ASI} of Figure 5.13 the rules of *r1*, *r2*, and *r5* of program Π_{fl1} are not necessary for understanding the outcome. Here the situation is complementary to ASI. As no knowledge is present that the products a and b are *cheap*, they are expensive. As a consequence, these products are not bought ($\neg buy(a)$ and $\neg buy(b)$) by the customer. The result of *ASII*, therefore, is: *ASII= {product(a), product(b), used(a), expensive(a), expensive(b),* $\neg buy(a)$*,* $\neg buy(b)$*}*.

Description Layer

The declarative nature of ASP is an asset for fastening the development process, but it is not able to point out which meaning is associated with each involved predicate for the human user. Therefore, an additional enhancement for the understandability of Π and its potentially hidden semantics is proposed by the definition of an own description layer. Descriptions are models describing the predicates of the used literals in more details. This layer is necessary as the other layers focus on the rule application or the rule design, but cannot describe the semantics and interrelations of used literals.

The notation can be based on UML Class diagrams [40]—and UML Object diagrams, respectively. The diagrams have to be extended to allow the versatility of predicates. Predicates can have different semantics. Generally, it is useful to differentiate between the usage of nouns (e.g. $car(X)$), adjectives/adverbs (e.g., *fast(X)*), and verbs (*buy(X)*). Nouns can be visualized like classes. They provide some attributes, relationships and a certain inheritance structure. Adjectives and verbs are classes without specific attributes, but can be in an inheritance relationship as well.

The detailed specification of the description layer is out of the scope of this thesis. The description layer can be seen as logical extension to the other layers. This layer is introduced as it is highly relevant for the design of Π to define and communicate the properties, behaviour and semantics of predicates used as ASP literals. However, the realization can be done in several ways, e.g., UML profile [40] or own models.

Optimization for Program Design

The introduction of constants in diagrams, increased the visualization complexity, as more elements are required to be integrated in the graph of Π. In the previous subsection, this increased complexity was tackled by defining different visualization levels based on the grade of abstraction, and by restricting the visualization to subsets of programs. However, for design-time visualizations of programs, all involved rules with all variables and constants should be accessible from a single artifact (the visual representation), which negatively influences the grade of complexity of this single artifact. Therefore, in this section a diagram is presented that is useable for design-centric approaches—placing the model as entry point of development (design-first)—rather than automatic generation for analysis. This transition from an analysis- to design-centric visualization is organized in two steps: (i) The extraction of the constant mappings and (ii) extraction of the variable declarations from literals.

Extracting Constant Mappings

The key difference to the approaches for analysis is the handling of variables and constants within the rule node. Within one rule a certain variable is unique. Beyond this rule, representing a scope of knowledge, the same variable name, e.g., X, can applied arbitrary times—they do not represent the same variable though. As these variables are unique within a rule, they can be attached to a rule to reduce the visualization effort, i.e., defining edges pointing to and from a variable. However, as the spatial close coupling of literals and the involved variables represents an asset for analysis-centric usage scenarios, the containment of variables in literal gateway nodes seems inappropriate for the program design usage. As a consequence the visualization for these two usage scenarios has to be different for variables.

For constants the knowledge boundaries are different than for variables. Each constant is a unique instance within an ASP program, e.g., *car1* or *expensive*. Constants relate to a variable in each eligible rule. Whenever a rule is applied, all constants corresponding to a variable used as head literal are added to this node being derived from nodes representing body literals of the same variable. This constant forwarding proceeds along the defined paths of a program Π.

As consequence, constants can be directly mapped to variables in the visualization, in lieu of mapping them to literals holding a corresponding variable.

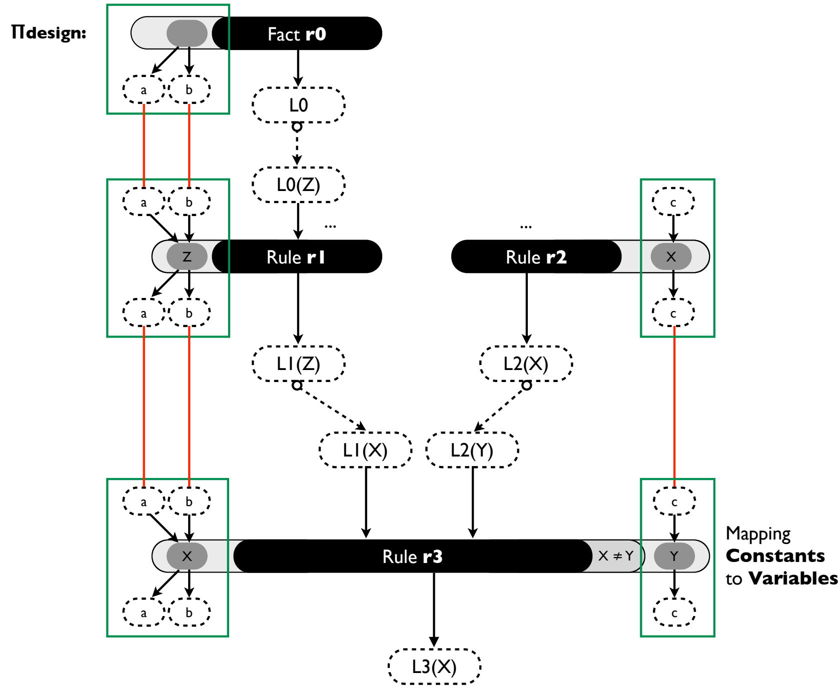


Figure 5.14: A design-centric model-based visualization of the ASP Π_{design}

A mapping of constants to variables is introduced in Figure 5.14 where three rules exist which describe the meta level structures of the ASP program Π_{design}—abstract deductions which are not based on the interrelations of constants. This concept is enhanced by *ConstantMappers* in the nodes of the rule—marked with a green border—which relate to the usage of (in-)/equalities. The *ConstantMappers* hold all involved variables as nodes ($variable$ $usages$)—marked with a grey background. For better visualization these mappers can be divided to be visualized on both sides of the rules. This division reduces the number of edges crossings in the graph. The variable nodes representing certain variables interact with a set of constants. Without previous definitions of constants corresponding to a variable within a rule, these interactions cannot be provided. However, from a given definition onwards a step-by-step interpretation is possible. This step-by-step interpretation is an eligible solution as it correlates with the rule-based step-by-step execution which is used in ASP. The fact in rule *r0* (aggregated from *r0 (a)* and *r0 (b)*) provides the initial mapping of the constants *a* and *b* to the literals being processable in abstract mappings. Root nodes as the mentioned fact *r0* do not hold any incoming edges—neither on meta nor on instance level. As facts do not make use of variables, the gateway node placed used for variables is unlabeled in Figure 5.14.

Our approach visually maps constants received via incoming edges, to rule-specific variables (cf. Figure 5.14). When a rule holding a set of constants on which a variable is dependent on

(usage in the body of the rule) is applied, these constants are forwarded to the head of the rule. As the interrelation of literals and applicable constants is already defined by the used variables, no explicit visual link of constants and literals is necessary. Such a behaviour can be seen in rule *r3* of Figure 5.14 where the constants *a* and *b*—represented as constant nodes—refer to the variable *X*. The involved constants interrelate. These interrelations represent relationships that can be of additive or subtractive nature (special variants of $E_{SR-Add.}$ or $E_{SR-Subtr.}$ edges respectively). These relationships are visualized with the help of undirected red edge. A subtractive nature means that constants referring to a variable are excluded from being used as constants of a specific variable of another rule. In particular, rule *r1* provides the constants *a* and *b*—locally mapped on the variable Z—in the state of literal *L1(a)* and *L1(b)* to the rule *r3*. In this rule these constants are locally mapped to the variable *X*. As a rule can consume from several rules, several edges can head to a constant node or exit from an outgoing constant node.

For the case of several literals referring to the same variable, several edges have to be headed to a variable gateway node. These edges are linked by gateway nodes representing the constants as input and output values. If a variable corresponds to a set of variables of other rules, a set of edges exist indicating these relationships. To simplify this handling an automated generation of edges between rules using constants is obviously useful, e.g., by integrating a solver. Alternatively, automated checking based on solvers could be established which visually indicate missing or impossible edges. The separation of operational information—although it is integrated in the same artifact—supports the ability for running automated procedures at a later stage than the development of the meta level structures and the definition of known root nodes.

Extracting Variable Declarations

In the approach of Section 5.4, the extraction of constraint mappings has already eliminated the necessity of constant nodes being added to literal nodes. In particular, this processing step has already initiated the shifting from a highly redundant rule- and variable-centric to a visually less complex variable-centric methodology. To continue this methodology shift, the variable declarations comprised in literals have to be extracted and combined with the literal definitions used for constants. This progress can again strongly decrease the visualization effort and therefore improve the ease of understanding of the graph.

This transition is achieved in three steps. First, the variables are placed as accessible nodes within rule nodes. Second, the variables from the literals are removed. Third, the gateway nodes for literals do not directly interact with the rule, but with the variable nodes held by a rule. Therefore, the edge from these gateway nodes point to variable nodes. These variable nodes represent the same variables that have been removed in the second step.

As this change can increase the edges pointing to a single variable node, it is essential to aggregate nodes in blocks where all incoming and outgoing edges can be shared by each participant of the aggregation. This is the case for constants that can be mapped to a variable. A constant mapped to a variable can be consumed and be used to produce outputs in the same way as all other constants referring to the same variable. Therefore, a *constant usage box* is applied bundling them graphically together. This bundling reduces the amount of edges pointing to the same variable. One jointly used edge as able to express the same behaviour for all used con-

stants. This improvement is highly necessary for programs holding a multitude of constants for each variable as this can happen, if real world examples are being solved by ASP.

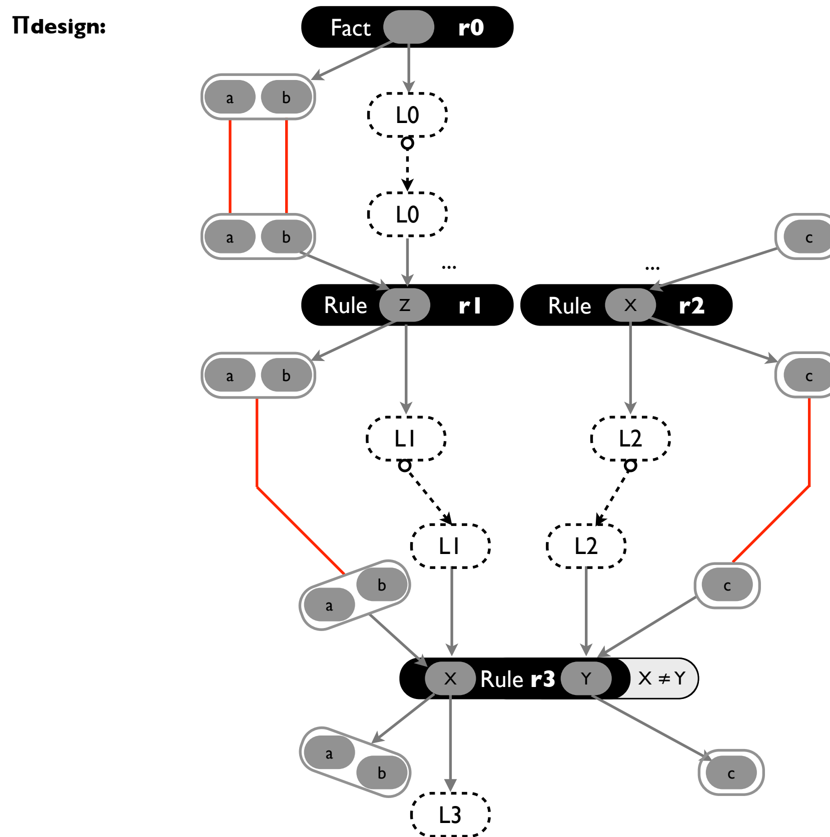


Figure 5.15: Mapping literals to variable nodes

In Figure 5.15, the program is adopted to this new visualization methodology. The rule *r0* is a fact that holds an empty variable box pointing to the meta structures (the gateway node representing the literal *L0*) and a *constant usage box* holding the constants *a* and *b*. This is equivalent to the textual notation of *L0(a)* and *L0(b)*. These values are than passed on to rule *r1* where *L1* is returned for these two constants. Rule *r2* acts similarly to rule *r1* mapping constant *c* using the variable *X* to the literal *L2*. The results of *r1* and *r2* are then consumed by the variables *X* and *Y* in rule *r3*. This example clearly demonstrates the visualization complexity reduction for adding further constants and literals in comparison to the initial visualization shown in Figure 5.14. Moreover, the visualization only follows one single methodology for both meta and instance level visualization. The two approaches in one visualization are joined to improve the consistency in modeling ASPs.

5.5 Programs of Literals with Unconstrained Arity

The expression capabilities of unary logic answer set programs Π_u are limited. Many typical algorithmic problems, e.g., the NP-hard Graph Coloring Problem [26], require the involvement of the neighborhood relationship, however. Such problems can have the following form: If *X* is a neighbor of *Y*, then X will satisfy the literal L_i. Such sentences cannot be expressed using the arity of one. Therefore, it is essential for the applicability of the presented concepts to support ASP programs with higher arity. For this purpose, the presented concepts have to be enhanced. From each literal node representing a certain literal, several variables need to be referenceable. This is achieved by allowing the usage of more than one E_D edge pointing from or to a gateway. As the ordering of variables of a literal, can influence the literal semantics, the ordering of outgoing or ingoing *D_Edges* has to be absolute. For example, the literal *better(X,Y)* states that *X* is better than *Y*. The literal *better(Y,X)*, however, expresses the opposite of *better(X,Y)* (cf. Figure 5.16). Relationships with undirected behaviour such as *equal(X,Y)* (cf. Figure 5.16) do not need orderings, but can make use of orderings. Although orderings could be implicitly expressed by some conventions or unique unlabeled nodes, this section enforces orderings to allow a clear correspondence between textual and visual representation of the same program Π.

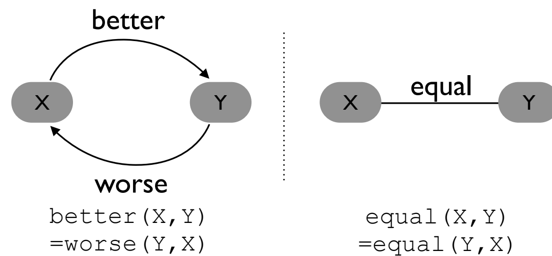


Figure 5.16: Directed and undirected literal behaviours

The ordering $O_{L_j}(A_1, \dots, A_k, A_{k+1}, \dots, A_N)$ where $O(A_{k+1}) > O(A_k)$ for the N arguments A_k of the Literal L_j can be visualized in several ways. First, *ordering boxes* (cf. Figure 5.17) can be used, which are placed within a gateway node of a literal. Each *ordering box* is labeled with an ordering value—an integer specifying its absolute ordering position within L_j. The variable to which or from which an E_D edge points to the *ordering box* (with a certain ordering value) of L_j is, therefore, equal to the variable at position of the ordering value ($V_{\text{ordering value}}$) of L_j in the textual representation. To simplify the visualization *ordering labels* (cf. Figure 5.17), labeling the E_D edges, can be used in lieu of *ordering boxes*.

In Figure 5.17 two examples are shown in the two different visualization approaches. The left example visualizes the literal $better(X, Y)$ (arity is two). It, therefore, requires the usage of two E_D edges—one points to the variables X and the other one to Y. For each used edge a *ordering box* or *ordering label* is necessary. The right example visualizes the literal $literal(X_1, X_2, \dots, X_k, \dots, X_N)$. As this literal has an arity of N, N edges and N *ordering boxes* or *ordering labels* are used.

Another necessary extension is the usage of constants—constants which do not correspond to any variable, but represent a fixed literal value—as arguments of a literal (cf. Figure 5.18).

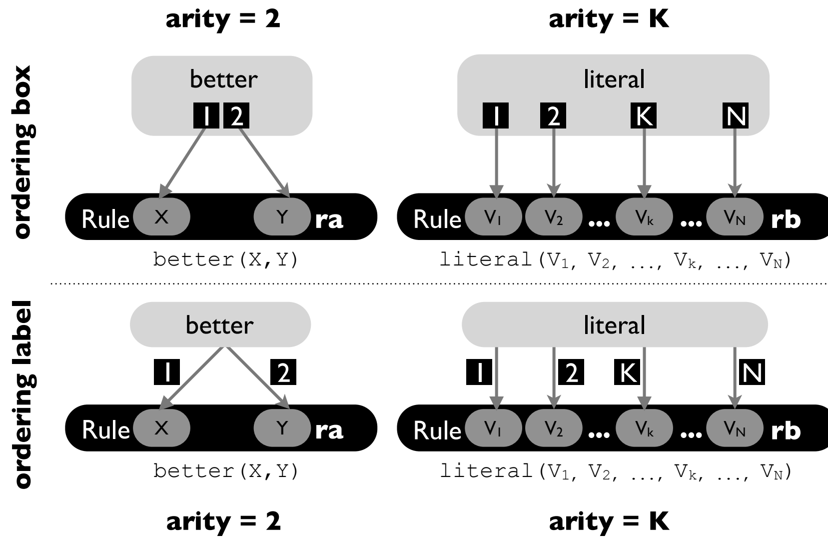


Figure 5.17: Graphical representations of answer set programs with arity of *two* and arity of K

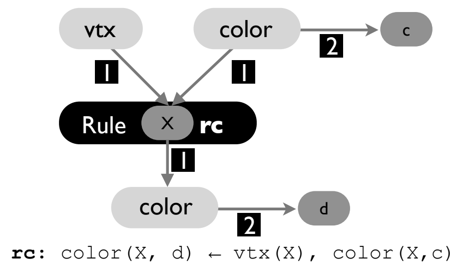


Figure 5.18: Using constants as arguments of a literal

In unary answer set programs the usage of constants are uncommon as ground literals would result, but in n-ary logic answer set programs it is often useful to use such constellations, e.g., in the Graph Coloring Problem the reduction to a number of colours also the definition of a rule for each case represented by a constant. In Figure 5.18, each vertex ($vtx(X)$) having the color (c) (*color(X, c)*), is recolored with *d* (X, d). The literal *color* has two arguments, which are label in the graph to correspond to the textual argument sequence—the vertex *X* is the first argument, whereas *c* and *d* respectively is the second argument.

5.6 Identification of Answer Sets

Answer sets are the results of a program Π. Directed graphs are used for the graphical representation of Π in the approach of this thesis. The computation of the answer sets based on the graphical representation can be provided by integrating solvers, e.g., DLV. This integration is considered to be future work beyond the general focus of this thesis. Nevertheless, for two

reasons it is important to state how a visual representation of answer sets is achievable. First, the computation of answer sets by solvers should be visually simulated in a step-by-step manner. Second, the visualization of these answer set calculations should be understandable by users.

This visualization therefore is undertaken by the help of defining answer set paths p_i of the graph. A precondition is that the visualized inferences (sender-receiver) relationships are valid—they have been automatically constructed or validated before, e.g., by solvers. The calculation of paths starts by the identification of root nodes. From each root node a set of paths can be constructed. A simulation of each answer set path p_i without solver can be undertaken by starting with a set of literals only containing the head literals of the root node, e.g., L_0 when L_0 is the root node of p_i. Special considerations are necessary for not-negated and disjunctions (cf. Section 9.2) which automatically split up this set into several subsets. Each path has a certain final node which does not hold any outgoing edges—nodes with no outgoing edges. Such final nodes can be calculated by searching for all nodes without outgoing edges. In particular, all constraint rules are final nodes as well as all other nodes without outgoing edges, which do not have a relationship to any constraint rule.

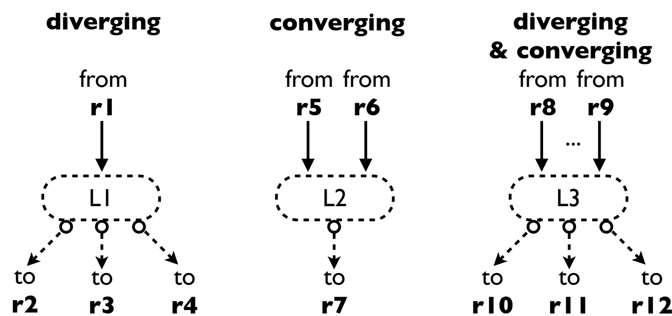


Figure 5.19: Converging, diverging and mixed edges

Each valid path from any root node L_0—no incoming edges—to any final node L_n is an eligible answer set path—if a literal tuple conforming to each node on the path is in the candidate answer set, it is a valid answer set. It is notable that nodes with several outgoing edges (diverging edges) from a node informally split a single path in several ongoing paths—this implies that computationally these two paths can be jointly simulated from the root of the path to the node from which edge diverge. Diverging rules can, but need not, be a result of disjunctive head literals. They can be result of several outgoing E_{SR} edges pointing to different applicable rules from a certain head literal of a predecessor rule. This behaviour is comparable with a choice-element where the path could continue. Answer set paths with the same final node form an answer set. This is especially important for converging edges of a node—two paths are directing to a single node L_n (cf. Figure 5.19 graphical representations of converging, diverging and mixed edges). From this point on the answer set paths are handled identically—such point is therefore named a join point in the simulation. Such join points computational aggregate the previously computed results in the simulation to a single answer set for which the computation is proceeded.

Definition 37 (Answer set path) *An answer set path of the graph* $G = \langle \mathcal{V}, \mathcal{E}, \mathcal{L} \rangle$ *is a path of the form p=*$v_1, \ldots v_n$ *where each* $v_i \in \mathcal{V}$ *(*$1 \leq i \leq n$*)* $v_1 \in root(G)$ *(root vertices of G) and* $v_n \in final(G)$ *(final nodes of G).*

By computing an answer set path from its root to its final node, all passed literals as vertices of the path are added to an intermediary result set. At each join point the computed literal sets of all related paths are merged. An example for a join point is given in Figure 5.8, in which $L5$ is dependent on $L1, L2, L4$. Each computed set of literals is regarded to be a candidate answer set—candidate answer sets that are answer sets without any validation against the constraint rules of Π.

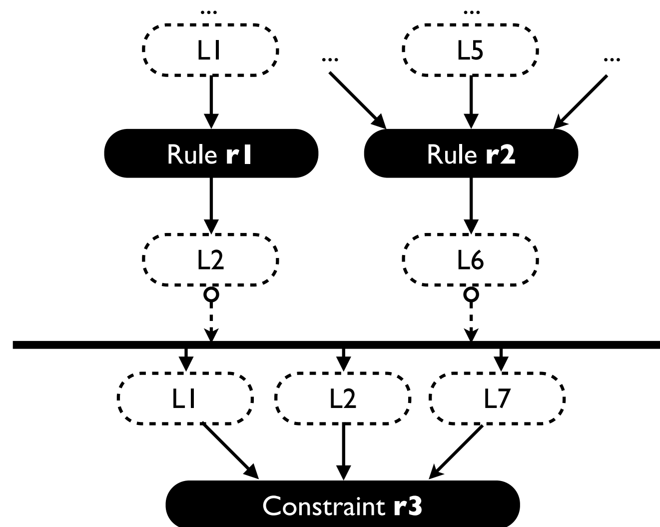


Figure 5.20: The answer set bus integrating constraint rules in the visualization

After the candidate answer sets are computed, a routine is necessary for identifying which candidate answer sets are valid answer sets (valid minimals models of *G*). Textually this is done by executing a program with a solver. This graph-based computation of answer sets, can be achieved by the integration of automatic routines such as solvers. In particular, it is important to integrate the computation of constraint rules and to validate, if the added edges visualizing inferences of ASP are specified correctly (no missing or wrongly set edges). For the integration of constraints of Π three variants are proposed: The *answer set bus*, *answer set containers*, and the *constraint rule integration* variant.

Answer Set Bus. The simplest variant is the visualization of the answer sets as horizontal line (the *answer set bus*). This line aggregates all paths together and links it with the defined constraint rules—constraint rules are always applicable to all candidate answer sets—cf. Figure 5.20 where such an answer set bus is positioned between the standard rules *r1* and *r2*, and the constraint rule *r3*. This aggregation implies that all head literals of each path pointing to this bus are candidate answer sets which have to be verified. If no connected constraint rule is

violated, a path represents an answer set. If it violates at least one constraint rule, the path cannot be returned as answer set. The simplicity of visualization is a great asset that is, moreover, strengthened by the high abstraction level. It can, therefore, provide a good overview of the relationships. However, to verify that p_i does not violate any constraint rule, all head literals of a path have to be known. These head literals can be computed and represent the candidate answer set of p_i. The answer set bus, however, does not provide any visual support for identifying these set of head literals of p_i. This can be regarded as disadvantageous as the developer has to compute such paths just-in-time to understand the full behaviour of Π.

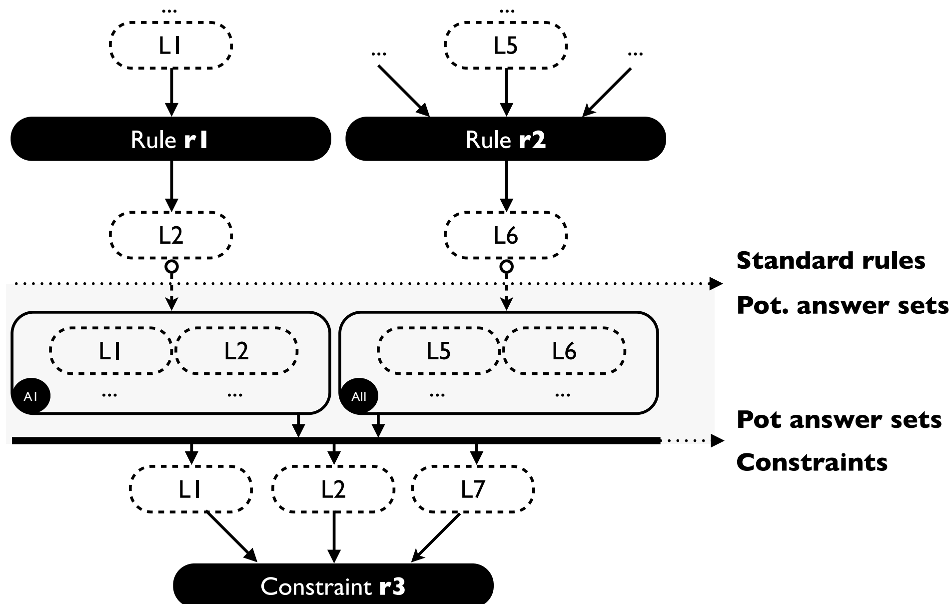


Figure 5.21: Computing answer sets for applying constraint rules

Answer Set Container. A possible extension of this approach is the visualization of particular answer sets. Answer sets are *containers* of a set of literals in the visualization (node visualized with black border containing a set of literal gateways). The containers have to be filled with the literals reached over the path directing to it (and joined paths). The filling process can be automatized by solvers. These candidate answer sets have to be linked with the constraint rules as described in (i). Without any computation or manual creation of the answer sets, the visualization does not provide a higher information level to the user. The visualization of literals comprised in all applicable answer sets reduces reduces the level of abstraction to some extend. However, it allows a visual identification which candidate answer set of Π does not violate any constraint rule and, therefore, is a real answer set. In Figure 5.21 two exemplary candidate answer sets (*AI* and *AII*) are described in an own layer by defining a box for each. All involved literals are shown in these boxes. From these answer sets all constraint rules can consume literals to satisfy their body literals. This visualization variant allows to identify that both described candidate answer sets are valid answer sets. *AI* satisfies *L1* and *L2* of rule *r3*, but may not satisfy

L7 (nothing indicates this satisfaction in Figure 5.21). *AII* does not even satisfy any of the body literals of *r3*. As no other constraint rule is defined for the answer set bus of Π, both are valid answer sets.

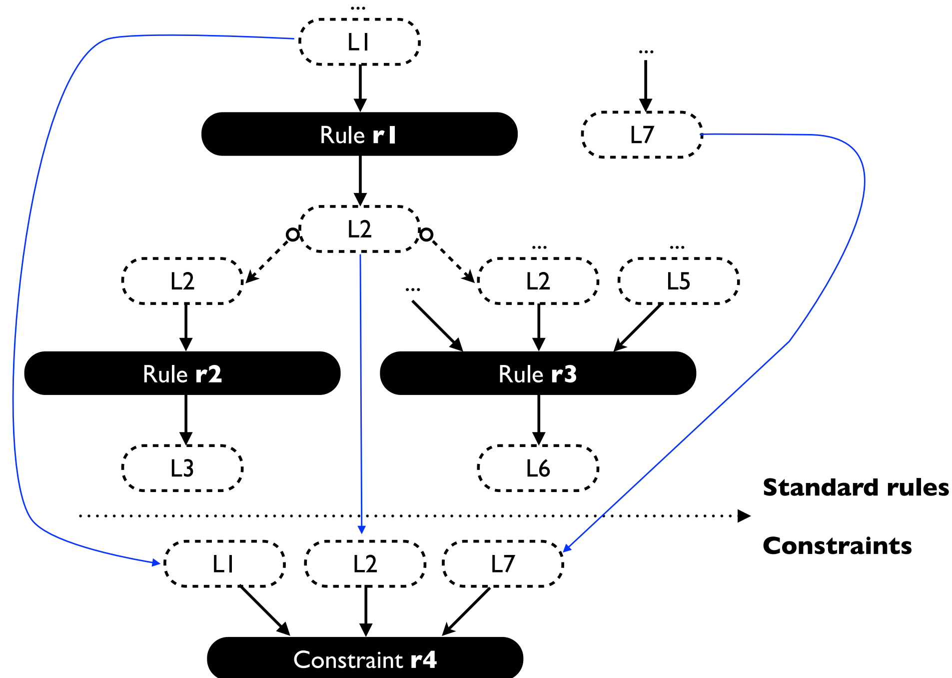


Figure 5.22: Visualization of all edges between constraint rule literals and other rule literals

Constraint Rule Integration. The *constraint rule integration* variant visualizes all edges from head literals corresponding to body literals of any constraint rule. As a consequence of the visual link between constraint rules and the occurrences of the involved literals, the resulting visualization is very extensive and visually complex. It allows the identification of origins of a constraint rule violation, which cannot be shown whether in the answer set bus nor in the answer set container visualization. As the origins of literals can be head and body literals of any layer of the graph, the edges often have to cross other edges or nodes. The more constraint rules are used and the more the constraint rules interact with rules, the more edges have to be placed on the graph. Intensive usages of constraint rules in combination with this visualization approach, could disallow the concentration on other rules comprising the inferences of the program. In Figure 5.22 all edges from the body literals of constraint rule $r4$ are visualized by blue edges—these edges are special consumption edges named *constraint edges*. These edges point to each existence of the stated literals in any other rule (which is no constraint rule itself). This could involve several edges from one body literal of *r4* to literals of other rules. This is not necessary for the body literal *L2* of *r2* and *r3*, as the rule $r1$ is the only origin providing *L2* for both rules. If there are more origin nodes for one literal L_i than consuming nodes on one level, it is advantageous to direct the constraint edge to the nodes of the consuming level (successor nodes).

All three variants have some advantages and disadvantages. The effort in computation is highest for the answer set container variant as an extensive simulation mechanism or solver has to be integrated. The effort in visualization is highest for constraint rule integration approach as the multitude of constraint edges have to be added from each head literal of the graph conforming to any body literal of a constraint rule. However, the expression power of answer set containers and the constraint rule integration approach is much higher than for the answer set bus, which does not provide a clear linking of origins of constraint violations and cannot simulate the computation effectively. However, for simplifying following figures of this section, the answer set bus will be mainly used.

5.7 Summary By Example

In the previous sections, the most essential features for visualizing ASP programs of any arity have been proposed. In the following example, a summary is given highlighting the most essential features proposed in this section.

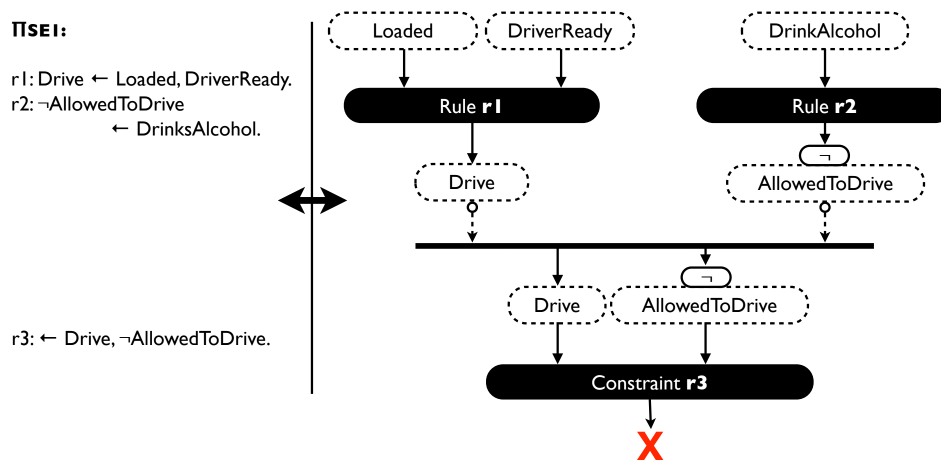


Figure 5.23: Visualization example of a propositional ASP program

First, only propositional atoms are used. This implies that no differentiation between operational information (instances) and rule design (abstract definition of rule behaviours) is necessary. This can be seen in Figure 5.23 where driving (*Drive*) is possible when the driver is ready ($DriverReady$) and the truck is *Loaded*. However, if the driver drinks alcohol, he is not allowed to drive. The constraint rule *r3* shows that a drunken driver cannot coexist with *drive* in an answer set, as this is legally prohibited—the boldly visualized *X* indicates a constraint violation. The program of Figure 5.23 it good example for highlighting the need for adding variables and constants in order to increase the applicability of the program Π_{SE1}, e.g., there could be several *Drivers*—one is drunk and cannot drive, but another one is not drunk and is allowed to drive.

Therefore, the usage of variables and constants is useful for differentiating between different instances. This is shown in Figure 5.24. In this figure, literals related to the same instance make use of the same variable. These variable usages represent the meta level visualization of the

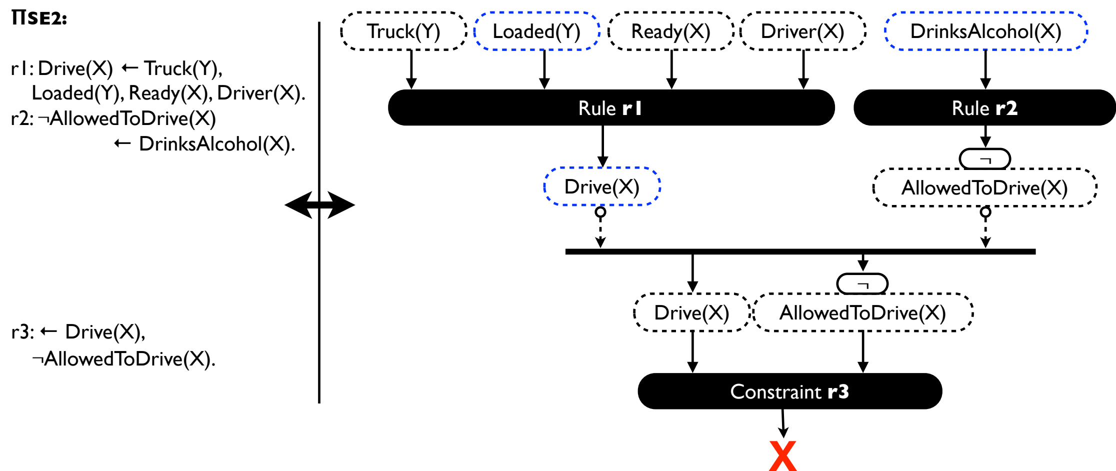


Figure 5.24: The meta level visualization of a unary logic ASP

unary logic ASP Π_{SE2}. The program Π_{SE2} is an extension of Π_{SE1} which introduces necessary variables and makes use of additional literals, e.g., *Truck(Y)*. Such literals are addable as unary logic programs are more expressive. The expressiveness of program Π_{SE2} can be increased, the higher arities can be visualized. Furthermore Figure 5.24 highlights, the boundaries of unary literals as such literals cannot describe interrelations in one literal, e.g., *Drive(X)* only describes that a particular *Driver* is allowed to drive, but it cannot express the interrelation with a certain *Truck* (his or her specific truck for example).

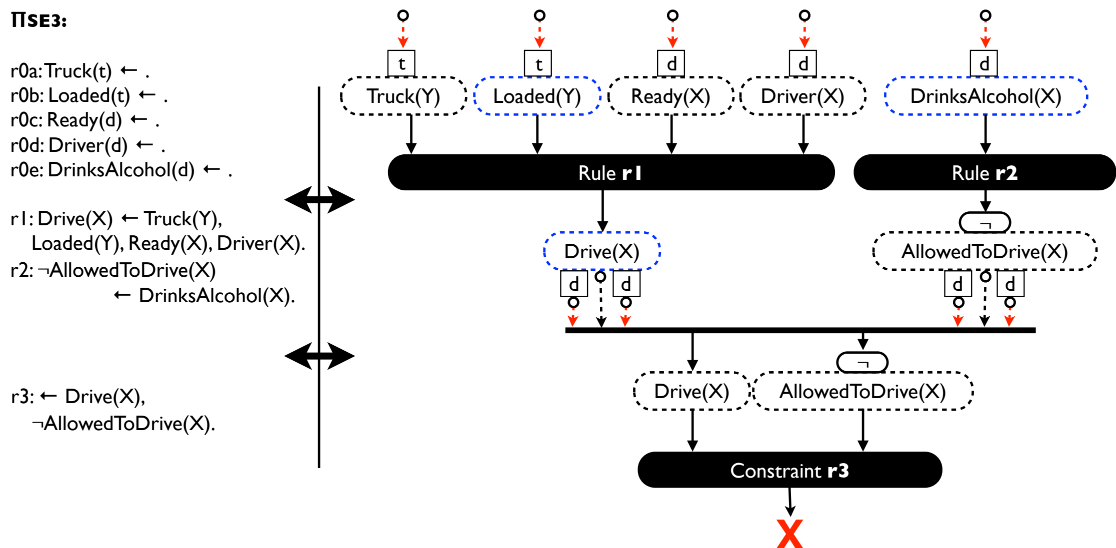


Figure 5.25: The instance level visualization of a unary logic ASP

As the meta level does not describe any constants (instances), the instance level answer set

visualizations are necessary. Such a visualization (the only one for this example) can be seen in Figure 5.25. This figure shows Π_{SE3} which extends Π_{SE2} by adding concrete facts. The graph only needs to visualize involved rules and their particular constant consumption and production for this particular case. In this example, however, all rules have to be visualized as all directly involve the result. In particular, no answer set is returned by Π_{SE3} as the constraint rule *r3* is violated by the driver *d* who is allowed to drive a particular truck *t*, but who is drunk as well. As this is prohibited in this example as it is prohibited by law of most countries, this answer set may not be returned.

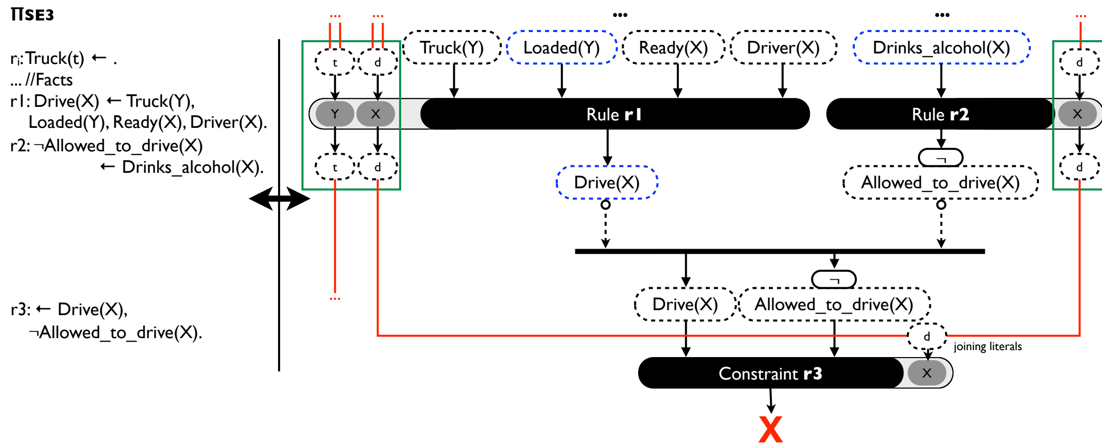


Figure 5.26: A design-centric model of the program Π_{SE3}

The stated unary models are more likely to be used for analysis as they hide some elements of Π and are very extensive for those elements of Π being displayed. The meta level hides the instance information and the instance level hides nodes not involved in particular computations. From a design-centric paradigm the model has to be able to express all information relevant for Π in a compact way. This is accomplished as visualized in Figure 5.26 where the literal arguments are encapsulated in the rule nodes, e.g., *X* and *Y* in rule *r1*. This separation of variables allows the rule-based mapping of constants to variables, e.g., *Truck(t)* is achieved by mapping the constant instance t to the variable *Y* in rule *r1*—the meta level literal *Truck(Y)* is, therefore, supplied with the instance of *t* like all other rules using *Y* as argument. This visualization variant allows the removing of the constant nodes placed at gateway nodes in previous analysis-centric approaches (cf. Figure 5.25). This reduction decreases the effort necessary to visually specify the program. However, this benefit is lowered to some extend as the number of constants is small. Furthermore, the visualization can be further optimized by reducing the variable declaration from each gateway node. This visualization can be seen in Figure 5.27. The behaviour is the same as in Figure 5.26, but the more compact visualization improves the ease of adding new body or head literals (gateway nodes) and constants.

The constraint rule visualization in rule *r3* of Figure 5.26 and Figure 5.27 is especially interesting. It might look as if the "variable to constants mapping" is not sufficient to highlight a certain violation. However, the variable nodes of the constraint rule *r3* is only supplied with

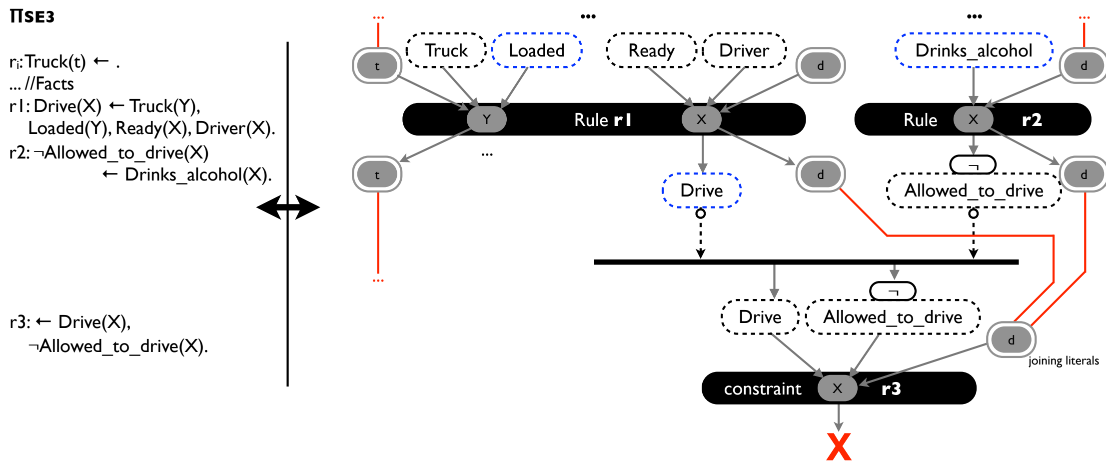


Figure 5.27: Extracted variable declarations in a design-centric visualization of Π_{SE3}

constants that satisfy all body literals of *r3* using the relevant variable. Others are not linked with this constraint rule and, therefore, do not indicate relevance for violation. Paths satisfying all literals of all variables, violate the constraint rule. For a better understanding of the satisfaction of involved literals, the edges are bound together in cases ($joining\ literals$ in Figure 5.26 for the binding of *Drive(d)* and *Allowed_to_drive(d)*) represented by *constant usage boxes* in 5.27. It is advisable to automatize or automatically check such linkings.

Another notable difference of analysis-centric instance level to design-centric models, is the necessity for design-centric models to visualize all specified structures of the program Π_{SE3} to provide a complete overview of the program behaviour.

CHAPTER 6

Realization

In this section, a prototype of the previously introduced modeling language for ASP is presented. Furthermore, the transformation from a model representation to the ASP code is achieved by proposing concepts for code generation. The visualization of ASP programs allows developers to concentrate on design issues rather than focusing on textual program syntax. In particular, we focus on giving a prototype implementing the most essential and recent concepts of the previous sections.

The presented realization approach of this thesis is named "**VI**sual **DE**sign and **A**nalysis **S**upport for **A**nswer **S**et **P**rograms" (VIDEAS ASP or briefly abbreviated VIDEAS). VIDEAS is Latin and can be translated to "You could see" (present subjunctive). VIDEAS ASP therefore states that you could see answer set programs Π (as specified in Definition 8) which underlines the focus of this thesis to highlight and visualize the essential elements of an ASP program Π. The used subjunctive expresses the possibility of gaining new insights into the behaviour of ASP programs.

6.1 Modeling Language

The proposed approaches are based on basic graph definitions (cf. Section 5.2). In this section, it is the intention to transfer this graph-based approach to concrete models and metamodels. This transition is eligible as models are based on graphs, and this transition is, furthermore, advantageous as models provide sufficient tool support, and defined abstraction levels (model, metamodel, metametamodel. . .), e.g., in the standardization of Meta Object Facility (MOF) [39] models by the Object Management Group[1].

The proposed modeling solutions are based on Eclipse Ecore[2] providing a technical realization of such abstraction levels for models. Ecore itself is fully integrated in the Integrated

[1]OMG information: `http://www.omg.org/`, last accessed: February 24, 2011
[2]Ecore information: `http://www.eclipse.org/modeling/emf/`, last accessed: February 24, 2011

Development Environment (IDE) Eclipse[3] which is used in version 3.5 “Galileo” by the Eclipse Modeling Framework (EMF)[4].

Environment & Boundaries

As this thesis has a primary focus of enabling design-centric program development, the design-centric approach allowing the visualization of inferences is prototypically realized within VIDEAS and documented in this and following sections.

The presented approach is a graphical visualization using models. Each model represents a particular program Π built for a certain problem specification. All of these models have to conform to a metamodel defining the VIDEAS modeling language. This metamodel is presented in Section 6.1 and is made accessible as graphical editor using Eclipse’s Graphical Modeling Framework (GMF)[5].

Metamodel

The metamodel has to contain a root class, which holds all other model elements. Each other class has to be transitively connected with the root class via containment relationships. This root class is called *DesignModel* in VIDEAS. Each class that is directly (distance of one) accessible to the root class is a class of special importance for the model, e.g., often reused class, a class with a major influence on the model definition, or a class referring to a multitude of other classes. In VIDEAS such classes are: *Rule*, *Variable*, *Constant* and *Literal*. The class *Rule* represents all rule nodes and handles variables, constants referring to these variables and literals specified over literals. *Variables*, *Constants* and *Literals* are root-accessible. Their usages in *Rules* are decoupled from their specification—this decoupling is realized by using own classes which are necessary with respect to the intended graphical visualization of explicitly showing the usages. Therefore, they have to be defined on the same level (root-accessible) as *Rules*. It is essential to reduce the amount of root-accessible classes to a minimum to improve the recognizability and ease of step-by-step defining models. Often the accessibility of reused classes can be preserved by creating an aggregation from a class that is hierarchically higher situated. Only if such a placement cannot be achieved or is not sufficient, classes should be made root-accessible.

The resulting metamodel of the defined modeling language, however, is very complex. As a consequence, it is divided into three main models. The model of Figure 6.1 provides an overview of the involved classes. Details to this model are then added in the models of Figure 6.2 and Figure 6.3.

All *Rules*, *GatewayNodes*, and *ConstantUsageBoxes* are subclasses of the abstract class *Node* (cf. Figure 6.2). The class *Rule* is a concrete class as facts, constraints, and standard rules are structurally similar to express. Therefore, each rule of any of these types can technically be changed to another type by adding or removing edges to *LiteralGateways*. The class

[3]Eclipse information and download: `http://www.eclipse.org`, last accessed: February 24, 2011

[4]EMF information and download: `http://www.eclipse.org/modeling/emf/`, last accessed: February 24, 2011

[5]GMF information and download: `http://www.eclipse.org/modeling/gmf/`, last accessed: February 24, 2011

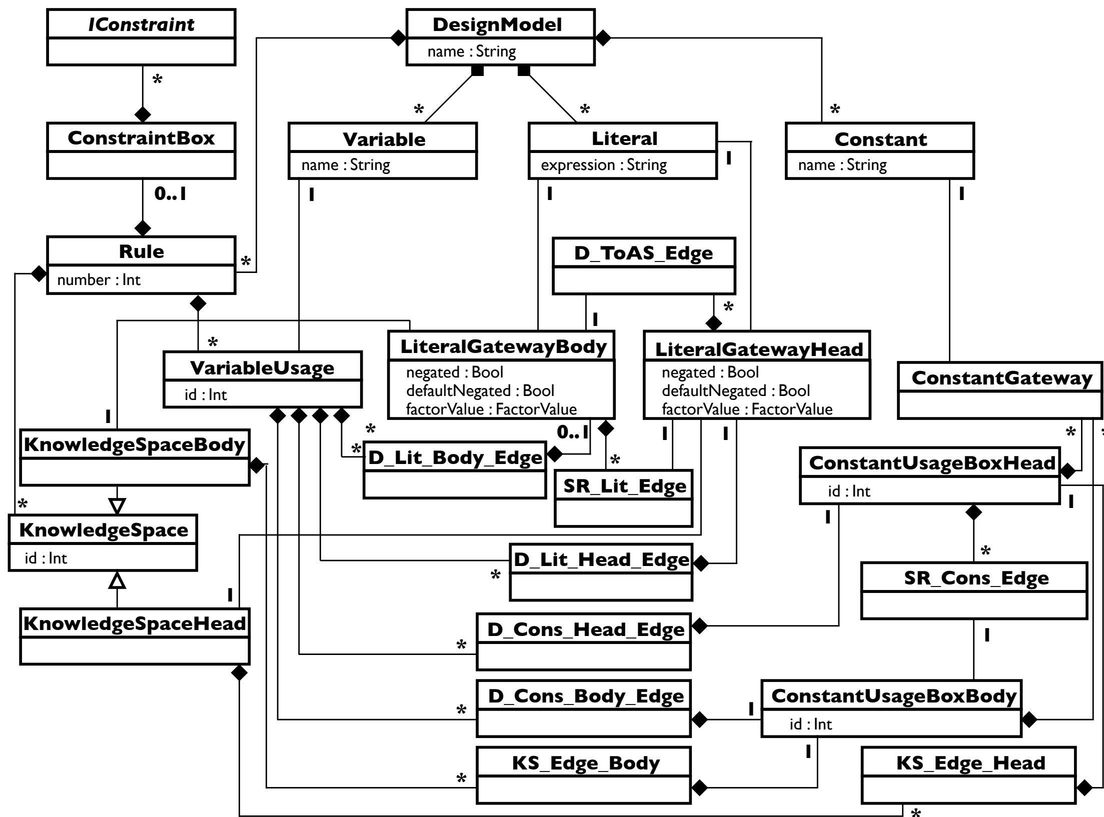


Figure 6.1: The simplified metamodel for the design-centric modeling language

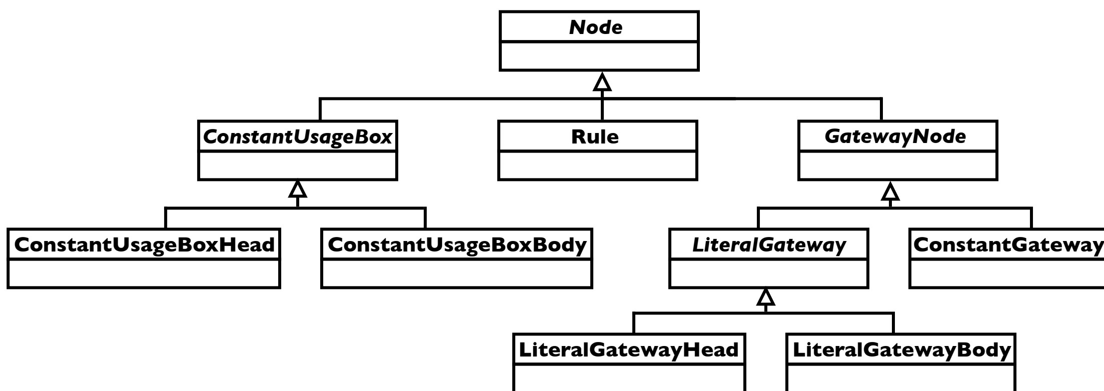


Figure 6.2: The inheritance relationships of the class *Node*

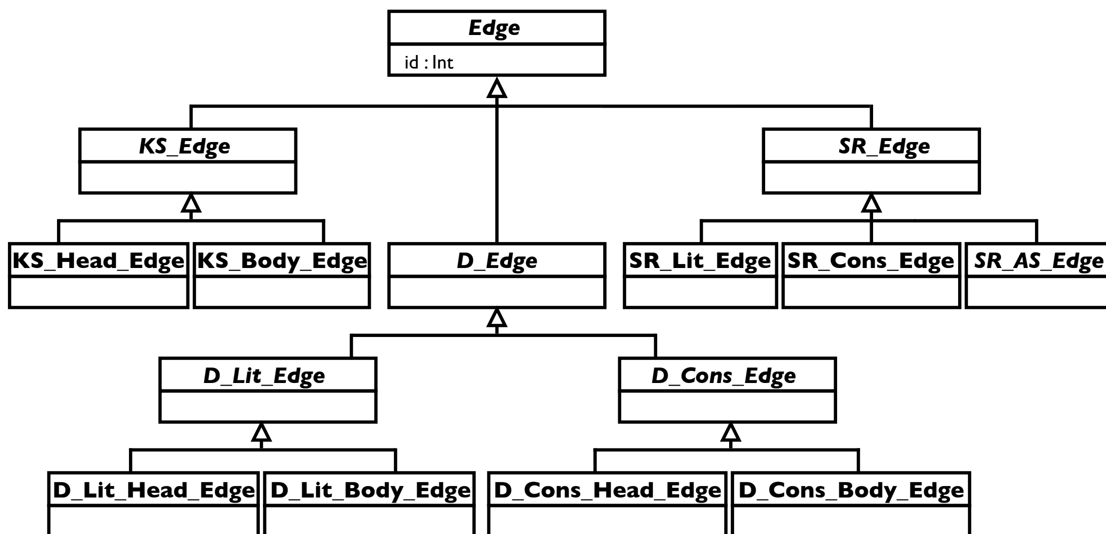


Figure 6.3: The inheritance relationships of the class *Edge*

GatewayNode is abstract as it can be differentiated between *GatewayNodes* representing *Literals* (*LiteralGateway*) and *Constants* (*ConstantGateway*). In contrast to the class *ConstantGateway* the *LiteralGateway* class holds to two concrete subclasses representing the specifics for the usage in the rule head and body—*LiteralGatewayHead* and *LiteralGatewayBody*. The class *ConstantGateway* does not require any specifics for the usage in the rule head or body, as all *ConstantGateways* are only addresses by the *ConstantUsageBox* holding them. The class *ConstantUsageBox*, therefore, has the two concrete subclasses *ConstantUsageBoxHead* and *ConstantUsageBoxBody*.

The inheritance structure of *Edges* is shown in Figure 6.3. There exist three major subclasses of *Edge*—the *KS_Edge* linking a *ConstantUsageBox* and *KnowledgeSpace*, the *D_Edge* representing direct dependencies to other items, and the (*SR_Edge*) sharing resources with other objects. These three subclasses are abstract classes and, therefore, cannot be instantiated. The *D_Edge* has the subclasses *D_Lit_Edge* and *D_Cons_Edge*. *D_Lit_Edge* expresses a direct dependency between a *VariableUsage* and a *LiteralGateway*. The *D_Cons_Edge* is used for defining a direct dependency between a *VariableUsage* and a *ConstantGateway*. Both classes are available as head and body subclasses as well. This differentiation is necessary to express the direction of the used *Edges*. The subclasses of *SR_Edge* are *SR_AS_Edge*, *SR_Lit_Edge* (abstract)—expressing the resource sharing between two *LiteralGateway*—and *SR_Cons_Edge* (abstract)—expressing the resource sharing between two *ConstantGateways*. The *SR_AS_Edge* are used to share *LiteralGatewayHead* nodes with constraint *Rules* consuming results from (*LiteralGatewayBody* nodes) other *Rules* (more details are presented later on). The two subclasses of *KS_Edge* are *KS_Head_Edge* and *KS_Body_Edge*. *KS_Head_Edge* links a *VariableUsageBoxHead* with a *KnowledgeSpaceHead*, whereas *KS_Body_Edge* links a *VariableUsageBoxBody* object with a *KnowledgeSpaceBody* object. For both *SR_Lit_Edge* and *SR_Cons_Edge* additively and subtractively variants exist.

Classes & References

In the metamodel of Figure 6.1 no differentiation is made between fact, constraint, and standard rules. This is not necessary as their visualization is not dependent on their number of incoming or outgoing edges (more details on *Rules* as constraints are discussed later on). However, their semantics is significantly different for the user. The class *Rule* holds *VariableUsages*, and a *ConstraintBox* via aggregations. *VariableUsages*—representing a variable within a rule—are the central element for binding *Literals*—in form of their usages represented by instances of *LiteralGatewayBody* and *LiteralGatewayHead*—to rules. The *ConstraintBox* is the container of all defined constraints. Such a *ConstraintBox* can hold a number of constraints (referring to the interface *IConstraint*) of different type. If no constraint exists, no *ConstraintBox* is necessary.

Constants. For each *VariableUsage* a set of *Constants* refers to the underlying *Variable*. All *Constants* eligible to be associated with the same *Variable*, are identically handled within the rule. These identical constants are bundled in *ConstantUsageBoxes*. This *ConstantUsage-Box* holds a set of *ConstantGateways*, which represent nodes referring to a single *Constant* each. The *ConstantUsageBoxes* themselves do not refer to any *Constant* in particular—they are only containers for gateways referring to *Constants*. It is highly important to state that the *ConstantUsageBoxHead* and *ConstantUsageBoxBody* addressing the same *VariableUsage* (*D_Cons_Edge*) and referring to the same *KnowledgeSpace* (*KS_Edge*) in the same rule, have to contain the same amount of *ConstantGateways* which refer to the same *Constants*. This can be addressed by defining them once for a *VariableUsage* and reusing them for the particular edges pointing from or to it. However, this separation hampers the forward development methodology in the model creation procedure. Without having defined the *ConstantUsageBox* before, it cannot be accessed when the developer is trying to define it. Furthermore, the preceding definition of the *ConstantUsageBoxes*, would require an additional class *ConstantUsageBoxesGateway*. This is necessary to redundantly highlight all incoming (body) and outgoing (head) *Constants* for each *VariableUsage* as it was proposed by the conceptual approach of previous sections. Without such a gateway, involved edges would have to point backwards to the same instance as used before which hampers the understanding of the model by the used top-down proceeding. However, this further separation resulting from additional gateways aggravates the intuitiveness of the modeling process. Therefore, in this thesis it is accepted that the *ConstantUsageBoxes* referring to the same *VariableUsage* could be defined inconsistently by the users. This drawback can be addressed by additional OCL constraints in the future. In particular, each *VariableUsage* can hold a set of incoming and outgoing *ConstantUsageBoxes* which are accessed by *D_Cons_Body_Edge* and *D_Cons_Body_Edge*, respectively.

Literals. The class *Literal* holds an attribute *expression*. The *expression* represents the String value of the *Literal* without defining the useable arguments, e.g., *Buy* in lieu of *Buy(X)*. These definitions are, therefore, independent from the gateway nodes using them. From the *VariableUsages* of a *Rule* the edges *D_Lit_Body_Edge*—the body edge consuming literals—and *D_Lit_Head_Edge*—the head edge producing literals—can be used to address and create *LiteralGatewayNodes*. In particular, one of those edges can only point to or from a single node. These *LiteralGatewayNodes* are those gateways only being able to refer to certain *Literals*. For this

purpose, each *LiteralGatewayNode* has to hold a reference to a single *Literal*. Therefore, these nodes are placeholders of *Literals* representing an application. Such applications are specific definitions for rules and their variables usages and are, therefore, dependent on their placement context. They hold further attributes describing this relationship—*negated* and *defaultNegated*.

Sharing Resources. The resources (literals or constants) referring to a certain *VariableUsage* can be shared with other rules. In particular, these resources are shared (additively or subtravtively) with other *GatewayNodes* (abstract) by using *SR_Edges* (abstract). From each *LiteralGatewayHead* a set of *SR_Lit_Edges* can be used which each point to a consuming *LiteralGatewayBody* of another rule. The same principle is applied for each *ConstantUsageBoxBody* which can share the constants with *ConstantUsageBodHead* instances of other rules. To reduce the amount of necessary edges, the sharing of single *Constants* using their *ConstantGateways* is prohibited. This addresses the problem that real world applications often tend to involve a multitude of *Constants* referring to the same *VariableUsage*.

Constraint Rules. The validation of constraint rules (*Rules* with constraint purpose) is achieved by using *SR_AS_Edges*. *SR_AS_Edges* point from *LiteralGatewaysHead* nods of standard *Rules* to *LiteralGatewayBody* nodes of constraint rules. Such *SR_AS_Edges* point from each *LiteralGatewayHead* on the graph to each *LiteralGatewayBody* of each constraint rule —referring to the same *Literal* as the *LiteralGatewayHead* node. To disallow the involvement of standard *Rules* as consumers of resources shared using *SR_AS_Edges*, two strategies are proposed. First, an own *Rule* class called *ConstraintRule* is created. This class again involves the necessity to create some further classes handling the special needs of the *ConstraintRules*, e.g., *ConstraintVariableUsages*, *ConstraintLiteralGatewayHead*, and *D_Cons_Head_Edge*. The effort of this strategy involves the creation of these classes and the references to these classes in the metamodel, as well as their integration in graphical editors in later sections. Second, the edge creation is automatized. Each rule without any head literals is automatically handled as constraint rule. Each *LiteralGatewayHead* node of standard rules is automatically connected with *LiteralGatewayBody* nodes of constraint rules referring to the same *Literal*. Optimally, even real-time modifications automatically lead to a revision of *SR_AS_Edge* usages. As the first strategy would optimally involve an automated *Edge* creation as well, it is beneficial to reduce the effort by applying the second strategy only. Therefore, the differentiation between different rule types is dispensable.

Graphical Editor

A graphical editor allows the graphical definition of models based on a certain modeling language. Each object in a model—based on the VIDEAS alphabet represented by the VIDEAS metamodel—is shown as graphical element[6] on the diagram[7] canvas. The available alphabet

[6]The term “element” generally refers to a graphical representation of an object based on a class of the VIDEAS modeling language in this section.

[7]The diagram in the context of GMF refers to the graphical editor of a certain model—in this thesis a model based on the metamodel of the VIDEAS modeling language

is defined by the classes of the metamodel representing VIDEAS. The editor handles the compatibility (grammar) of these items in the model definition by allowing and disallowing certain placements on the canvas. The compatibility is derived from the specified references in the VIDEAS metamodel. For each of those concrete classes (items on the canvas) a certain visual representation has to be specified. In particular, the visualization has to allow the distinction of the used classes—especially the distinction of classes representing edges and nodes have to be made. Such a distinction can mainly be achieved by using different shapes and colors.

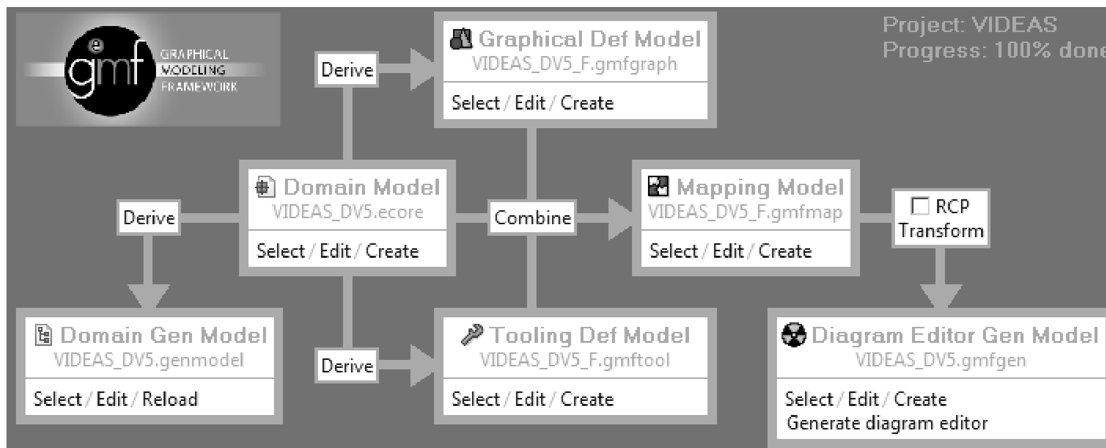


Figure 6.4: The *GMF Dashboard*[8] guiding the creation of the graphical editor

The graphical editor for the design-centric modeling language VIDEAS is created by using the editor functionality of Eclipse GMF. With this help, the *EMF Generator Model* (*genmodel*) is created from the metamodel (defined in EMF). This is a "data model"[9] used as intermediary format. The *genmodel* allows the generation of the code classes from the modeled classes in the EMF metamodel. Furthermore, the *genmodel* is a prerequisite for creating a graphical editor with GMF. Even the code that can be generated from this *genmodel* is necessary to run the GMF editor. To establish an own graphical editor, the *GMF Dashboard* is provided by GMF. The *GMF Dashboard* (cf. Figure 6.4) guides the creation of the graphical editor by assisting in the creation and combination of the necessary files. Necessary files are the *Domain Model* (the VIDEAS metamodel), the *Domain Gen Model* (the produced *genmodel* from the metamodel), the *Graphical Def Model*, the *Tooling Def Model*, the *Mapping Model*, and the *Diagram Editor Gen Model* which is the overall outcome. After integrating all of these model files, the graphical editor may be used within the Eclipse environment.

[8]Information on GMF and GMF Dashboard: `http://www.eclipse.org/modeling/gmp/`, last accessed: February 24, 2011

[9]Data model information: `http://wiki.eclipse.org/Graphical_Modeling_Framework/GenModel/Hints`, last accessed: February 24, 2011

Adopting the Metamodel to GMF Requirements

The creation of a model using a graphical editor has a higher requirement of root-accessible classes than non-graphical models. Root-accessible classes in graphical editors are classes that can be placed on the canvas, which represent the root class. Therefore, all classes, which have to be directly placeable on the canvas, have to be made root-accessible. The particular difference to non-graphical models can be found in the transitive placement of graphical elements, e.g., *Edges*. An *Edge* is typically in an relationship to a source element, and points to a target element (another relationship). As an *Edge* is neither visually placed within another graphical element (compartment). The source or the target of the *Edge*, nor automatically placed on the canvas when another item is added, it has to be made root-accessible in the metamodel in order to allow an independent canvas placement. Non-graphical models, however, can construct such object sequences by using a series of containments pointing from one object to its followup (based on containments)—a direct relationship to the root-class need not exist. Therefore, some classes have to be made root-accessible to support the graphical placements. Mostly this can be achieved by declaring an inheritance relationship to the classes *Edge* or *Node* which are root-accessible themselves. *KnowledgeSpaces* are rather virtual constructs that can be automatically generated to highlight the accessibility of certain nodes of the class *LiteralGatewayHead*. They are, therefore, not applicable to be defined as classical node and are directly made accessible (containment) by the root class.

Another extension is necessary to improve the maintainability of *Edge* definitions in the involved GMF models—especially the *Mapping Model*. GMF requires each *Edge* (defined as $Connection$) to provide the reference to a source and a target element—the endpoints of the *Edge*. Both references point from the *Edges* to the involved endpoints. These references have to be defined—if not already existing for the particular class—and named properly. The naming is essential for the maintainability of the GMF model files. Therefore, all these references are named “source” or “target” respectively. By using these two references, the transitive containments from source elements via *Edges* to the target elements are redundant. As *Edges* can be used by more than one source class and *Edges* can point to more than one target class, this naming convention could not be adopted to each of the subclasses of *Edge*. Therefore, several source and target references could exist with similar naming. These relationships, therefore, have to be interpreted in the GMF models. As the regeneration of GMF models from particular sources can lead to the removal of such definitions, the necessary mapping has to be renewed. To avoid the difficulties identifying the correct source and target element, subclasses of existing *Edge* classes are created that only have to hold a single source and a single target reference—they, therefore, represent a single usage purpose. This breakdown of existing usage purposes, simplifies the GMF model definition and maintenance.

Moreover, GMF in the current version has some visualization limits, which disallow the usage of boolean types, i.e., *negated* and *defaultNegated* in the *LiteralGateway* of the VIDEAS metamodel. Consequently, boolean values are expressed by enumerations (*Enum*) of strings in graphical editor. The *Enum Negation*—used for the attribute *negated*—holds an empty String (*negated* is false) and “neg” (*negated* is true) as *Enum Literals*. The *Enum DefaultNegation*—used for the attribute *defaultNegated* holds an empty String (*defaultNegated* is false) and “not” (*defaultNegated* is true) as *Enum Literals*.

Graphical Def Model

The *Graphical Def Model* is used for defining the graphical appearance of the model objects. For each graphical element (visually delimitated from other elements by its shape, color, border, usage within the graph, etc.) a figure is defined. Typically for each concrete class a figure is created—especially when this model is generated from a metamodel—which can be adopted to its special needs. However, a unique figure definition for each class of the metamodel might not be necessary, as functional similar classes may require the same appearance, e.g., *ConstantUsageBoxHead* and the *ConstantUsageBoxBody* have the same appearance, but have different relationships. The *Graphical Def Model* supports a wide range of supported shapes (e.g., rectangle or ellipse) for connections as well as for classes. It, furthermore, allows the usage of elements as *Labels* and *Compartments*. If the standard adjustments are not sufficient, even own classes referring to predefined shapes can be created and used.

In VIDEAS, mainly rounded rectangles are used to express nodes. Elements being used as containers for other elements (*Compartments*) are visualized with a light background color. The only exceptions are *Rule* elements—the most essential element on the canvas—which are, therefore, marked with a background in deep grey. Basic elements—e.g., *Literals*—are expressed by simple elements (typically as square rectangles) to allow better differentiation. Polylines are used to express connections (*Edges*). In particular, direct dependencies are represented by directed *Edges* and indirect dependencies are represented by visually undirected *Edges* (no arrows are placed on the graphical shape of the edge).

Typically, names or useful identifiers, e.g., the identifier of *Rule*, are used as *Labels*. Moreover, it is essential to use the attributes *negated* and *defaultNegated* of *GatewayNodes* as *Labels* to allow easy recognition of possible rule applications.

Tooling Def Model

The *Tooling Def Model* is necessary to define which elements are placed in the tooling palette of the graphical editor and which ones are placed in the menu bar. VIDEAS only makes use of palettes at the moment of writing. Additional elements can be easily added to the palette or to the menu bar in extensions. The defined tools in VIDEAS are grouped in blocks. Such blocks help to organize the available elements in such a way that they are easily findable. In particular, four tool blocks are defined (cf. Figure 6.5): (i) Basic elements, (ii) rule-oriented elements, (iii) edges, and (iv) refinement elements. The block (i) represents the most essential elements which are prerequisites for all other elements. In VIDEAS, this involves the classes *Variable*, *Constant*, and *Literal*. All of these classes are not directly used to specify the flow of rules. All classes standing in a direct and close relationship to *Rules* are placed in block (ii). All available concrete *Edges* are placed in block (iii). The tool block (iv) concentrates on features which are not used by the majority of diagrams representing Π Such features, e.g., *OCLConstraint*, *InequalityConstraint* and *InequalityConstraint*, are meant as refinements of the graphical models.

The *Tooling Def Model*, furthermore, allows the usage of other names than defined by the metamodel class names. This is not used for VIDEAS, but can help to optimize the class names for their visualization, e.g., abbreviations, and can be used for modifications in the future.

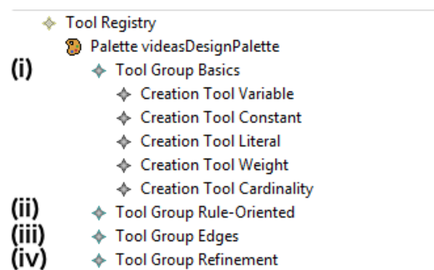


Figure 6.5: The VIDEAS *Tooling Def Model*, with collapsed and uncollapsed tool groups

Mapping Model

The *Mapping Model* (cf. Figure 6.6) combines the visualization defined in the *Graphical Def Model* with the palette definitions of the *Tooling Def Model*. All previously discussed models are included in the *Mapping Model* by using certain references. The main mapping definitions of this model are, however, undertaken in the block *Mappings* which allows the placement of two different mappings: (i) *Top Node Reference* and (ii) *Link Mapping*. All classes being represented as own nodes on the canvas are declared as *Top Node Reference*. All *Top Nodes* are, therefore, root-accessible classes that are chosen from the referenced metamodel (VIDEAS metamodel). All classes that are not root-accessible, can only be used as *Compartments* or *Labels* of *Top Nodes* by adding a child reference. Such references point from *Top Nodes* to elements being accessible from this *Top Node*. *Compartments* allow the placement of elements within the boundaries of another element—such elements in VIDEAS are *ConstantGateways*, *VariableUsages*, *ConstraintUsageBoxes*, and all three classes referring to the interface *IConstraint*. *Labels* allow the presentation of some value directly visible on the canvas—e.g., *Name* of the class *Rule*. Each added *Label* increases the visualization complexity. Consequently, VIDEAS focuses on some labels being essential for the recognition of the program only, i.e. mainly name attributes. *Compartments* and *Labels* can only be declared as child elements of *Top Nodes*. All *Edges* in VIDEAS are declared as *Link Mapping*. For such *Link Mappings* a source and a target reference has to be defined.

Required labels have to be added to the *Top Node Reference* or *Compartment* definition in the *Mapping Model* as well. They have to combined with their visual representation in the *Graphical Def Model*, but need not be linked to any definition of the *Tooling Def Model*. A special case of such labels represents the labeling of referred object, e.g., a *Variable* from a *VariableUsage*. Such labelings are not supported by GMF and, therefore, are not addable to the *Mapping Model*. They are realized in a self-created *custom* project which overwrites the visualization of the involved nodes. The *custom* project is necessary to allow modifications of the *Mapping Model* without having the need to add this functionality by hand again.

The *Mapping Model*, moreover, offers some further functionalities like reuse of graphical and tooling definitions for other mappings. For example, if an item cannot only be used as *Compartment* of several other elements, but it can also be placed as *Root Node*, several entries for the same visualization and tooling element can be created providing this functionality. For example *LiteralGateways* nodes can directly be placed on the canvas or be placed within the

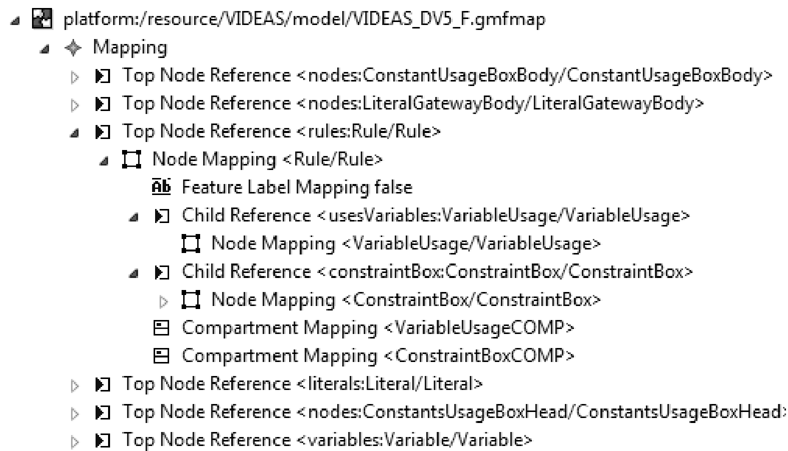


Figure 6.6: An excerpt of the VIDEAS *Mapping Model*

boundaries of a *KnowledgeSpace*, which differ in some advanced features as presented in Section 9.2. Such variations can be easily achieved by removing the child access (for a particular class) for the *Top Node* definition in the *Mapping Model*. The *Mapping Model*, furthermore, allows the exchange of used graphical visualizations, tooling definitions, or related classes. If elements are not listed in the *Mapping Model*, they are not useable in the graphical editor.

Diagram Editor Gen Model

The outcome of the process guided by the *GMF Dashboard* is stored in the *Diagram Editor Gen Model*. Before the generation of this model can take place, all previously discussed models have to be selected. By varying some of the referenced models, several different outcomes can be produced and stored as *Diagram Editor Gen Model*. The resulting model is used as intermediary format for generating the code for the diagram editor. As the *Mapping Model* already unifies the definitions of all models in one model representation, no configuration is typically necessary in the *Diagram Editor Gen Model*.

From this model automatically the diagram code (the main code for the graphical editor) can be generated. It is completed by the code being generated from the *genmodel* which holds the classes representing the model elements themselves (e.g., *Rule*). Afterwards the resulting projects can be exported as Eclipse plugins and integrated in the Eclipse IDE. For testing purpose the projects can be started as *Eclipse Application*.

VIDEAS Editor

After integrating the VIDEAS plugins representing all generated code of the EMF and GMF models, VIDEAS can be used. The VIDEAS diagram (the graphical model in GMF) can be created by adding a new file to an existing (or newly created) project.

The VIDEAS graphical editor by default shows the *Toolbar* on the right, and the *Canvas* on the left side (cf. Figure 6.7). The blocks in the *Toolbar* of the VIDEAS graphical editor are

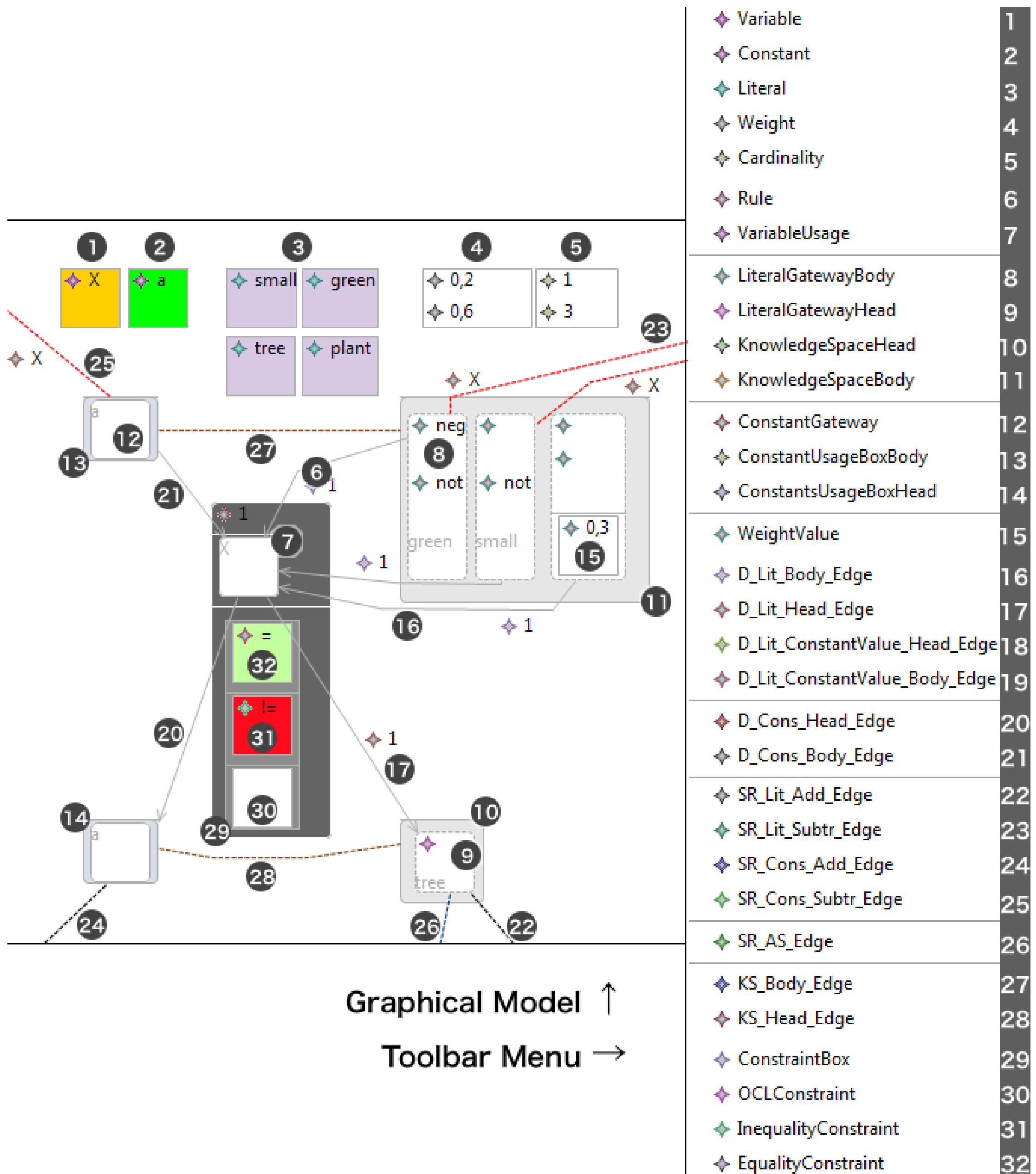


Figure 6.7: An exemplary usage of the VIDEAS graphical editor

visualized by bigger row margins—e.g., (5) to (6). Each block holds *Palette separators* (solid lines) to further structure the toolbar in semantical groups—e.g., (8) - (11). Each listed element in the *Toolbar* is used and numbered at least once on the *Canvas* in Figure 6.7. The numbers on the *Canvas* correspond to the numbers next to the *Toolbar*. These numbers are used in this Figure to highlight the actual visual representation that could be achieved for each element in GMF. The graphical editor as shown in Figure 6.7 is, moreover, more colorful than presented in the conceptual approach (cf. Figure 5.27). Some further deviations exist as GMF sets some visualization limitations, e.g., labels to referenced objects (custom visualization) cannot be recognized on strong background and, therefore, require white backgrounds.

The GMF editor automatically supports *Connections* defined as *Polylines* (representing *Edges*) that connect vertexes—(16) to (28). The naming of the elements—(1) to (32)—in the *Toolbar* is the same as the concrete classes they represent. The elements *D_Lit_ConstantValue_Head_Edge* (18) and *D_Lit_ConstantValue_Body_Edge* (19) are not shown in Figure 6.7 as they have the same appearance as the edges from (16) to (21).

6.2 Code Generation

From the graphical model representation of program Π (defined by using the graphical editor of VIDEAS) a code generation mechanism is necessary to execute Π with a certain solver S. For this purpose, the Eclipse project Model To Text (M2T)[10] is used. In particular, the template language *Xpand*—which is assisted by *Xtend* and *Check* files—is used. One transformation (code generation) file (*UnaryTemplate*) is established, which defines the output created in the transformation process. The *UnaryTemplate* file can transform answer set programs with at an arity of one including the usage of negations. Additionally, another transformation template (*NaryTemplate* is provided which is capable of generating the basic code structure for programs of any arity—fact rules as well as advanced features of Section 9.2 are not supported. The restriction of the n-ary generation is related to technological boundaries which increase the effort of building efficient string chains of the arguments of a literal (such arguments represent cross-products of the listed *ConstantGateways*). In VIDEAS, for each model a new file is created with the file ending *.dlv* as the textual notation is based on DLV (some advanced features are added which are not supported by DLV). *Check* is used to ensure that the model being transformed fulfills certain constraints, e.g., a *Rule* cannot consume (*LiteralGatewayBody*) from its own *LiteralGatewayHead* nodes. The transformation is executed by a generator file calling the *main* method of the *UnaryTransformation* with a certain input model—a model based on the VIDEAS generator model, e.g., created by the VIDEAS graphical editor.

The transformation process is *Rule*-centric. Each node connected to a *Rule* (via an *Edge*) or contained elements of a *Rule* are transformed. All other elements are not relevant for the code generation. Elements not being transformed can be necessary to support the understanding of the behaviour of Π or can be redundant—unbound elements which do not support the understanding and have no influence on Π's behaviour. As a VIDEAS *Rule* is powerful, but complex

[10]Information and download: `http://www.eclipse.org/modeling/m2t/`, last accessed: February 24, 2011

(many incoming and outgoing edges), (i) specific code generation procedures for *fact* and *standard Rules* are defined, and a separation of the code generation for head and body literals of a *Rule* is required. (i) Fact *Rules* cannot be transformed like standard *Rules* as facts do not hold any *LiteralGatewayBody* nodes. Variables pointing to *LiteralGatewayHead* nodes, therefore, are unbound. Therefore, for each *LiteralGatewayHead* for each referenced *ConstantGateway* a constant-specific head literal is generated in the textual representation. Standard *Rules*, however, are responsible for a set of constants and, therefore, involve *Variables* instead of particular UnaryTransformation*Constants*. (ii) Head and body of *Rules* are symmetrically defined around a *Rule*. However, the involve specific head or body specific classes which have to be accessed and transformed. The produced code, therefore, is similar, but the processing requires a differentiation. The separation of code generation around the dimension of head and body literals is even more necessary for providing the opportunity of adding specific transformation strategies, e.g., restrictions, in the future. All four blocks obtainable from these two dimensions are technically established by two *Xtend* files: *Extensions* and *GeneratorExtensions*. *Extensions* are used to simplify the access to certain model elements, e.g., accessing the container element of a *Rule*—the *DesignModel*. The *GeneratorExtensions* are responsible for supporting the business logics for often reused code sections. Such often reused code sections are defined in the *GeneratorExtensions* and are then called at multiple locations in the *MainTemplate* file. This separation improves the readability of the transformation code and is, therefore, essential for the maintainability of the generator.

Moreover, the generated code is normally not well-formatted. For this purpose, so called beautifier are used that remove unnecessary white spaces and line breaks. As M2T out of the box only provided a beautifier for XML and for Java, a answer set program specific beautifier had to be written. VIDEAS, therefore, provides a simple beautifier called *VIDEASBeautifier*. This beautifier removes all line breaks which are no direct successors of dots and tokenizes each such a way hat each block is only separated by a single whitespace. A direct integration of the *VIDEASBeautifier* in *M2T* was not implemented in the initial version of the VIDEAS code generator. The resulting code, beautified with the *VIDEASBeautifier*, is executable with *S* and returns answer sets of Π.

6.3 Summary

The proposed approach is named VIDEAS. For this approach two main prototypical contributions were given: (i) The graphical model editor, and (ii) a code generator. (i) The basis for the graphical model editor is defined by the VIDEAS metamodel. Several features are integrated in this metamodel such as the support of programs using logic of unconstrained arity, and some additional features presented in Section 9.2. (ii) From graphical models defined by the graphical editor code can be generated using the VIDEAS code generator which is optimized for DLV. In particular, two generators are given: One is optimized for unary logic programs providing the advanced features of the VIDEAS metamodel, the second represents a generator of the basic elements of n-ary logic programs.

CHAPTER 7

Evaluation

This section aims for evaluating the previously presented approach and its prototypical realization by examples. For this purpose, two answer set programs are given as exemplary usage scenarios—a unary logic and a binary logic program. Both represent typical answer set programs of the ASP literature. The binary logic example, however, is used to involve the suitability for more sophisticated real world scenarios which often cannot be expressed with the limited capabilities of unary logic programs. These examples are both visualized in VIDEAS and then evaluated in these two categories: (i) Visualization and (ii) generated code.

7.1 Usage Scenarios By Example

Two typical examples are chosen to allow the evaluation of the introduced approach. First, a simple unary logic program (named *Penguins-Cannot-Fly* problem in this thesis) is given which answer the question with animals can fly. Especially, the non-flying penguins are considered as special case. The second example (*3-Coloring-Problem* [26]) represents one of the first NP-hard problems that has been published. This example, moreover, contains binary predicates.

Unary Logic

The simple *Penguins-Cannot-Fly-Problem*—as it is named in this thesis—is simple problem that tries to identify which *birds* can *fly*. This exemplary program ensures that *penguins* are *birds* that cannot *fly*, but all other *birds* are able to *fly* (this is naturally not transferrable to the biological context). This behaviour is achieved in three rules (two facts and two standard rules). The facts are used to setup the used interpretation. In particular, the two *birds tweety* and *pingu* are considered. The constant *pingu* represents a *penguin* which, therefore, cannot *fly*. The standard rule, however, provides the main behaviour of this program. Only *birds* are considered—*bird(X)*—which are not knowingly *penguins*—*not penguin(X)*. These two conditions are represented as body literals. For all *birds* which are no *penguins* the rule is applied and the head literal *fly*—*fly(X)*—is added to the answer set.

The stated solution for the *Penguins-Cannot-Fly-Problem* is based on the answer set program of Equation 7.1. Each application of facts for a *Constants*, is represented as own rule. The more constants are added to this solution for the *Penguins-Cannot-Fly-Problem*, the more textual rules are necessary. As two *Constants* for *bird* are used, the two facts *r1* and *r2* are necessary.

$$\Pi_{7.1} = \begin{cases} r1 : bird(tweety) & \leftarrow . \\ r2 : penguin(pingu) & \leftarrow . \\ r3 : bird(X) & \leftarrow penguin(X). \\ r4 : fly(X) & \leftarrow \textit{bird(X), not penguin(X).} \end{cases} \tag{7.1}$$

Graphically, this problem is shown in Figure 7.1, which involves the following nodes and edges. For the sake of readability the figures have been digitally reworked. The rules *r1* and *r2* can be visualized as single fact rules which hold *ConstantGateways* for each matching *Constant—tweety* for *r1* and *pingu* for *r2* (cf. Figure 7.2). The graphical notation, therefore, only requires the assignment of a new *ConstantGateway* in lieu of additional independent rules as in the textual representation.

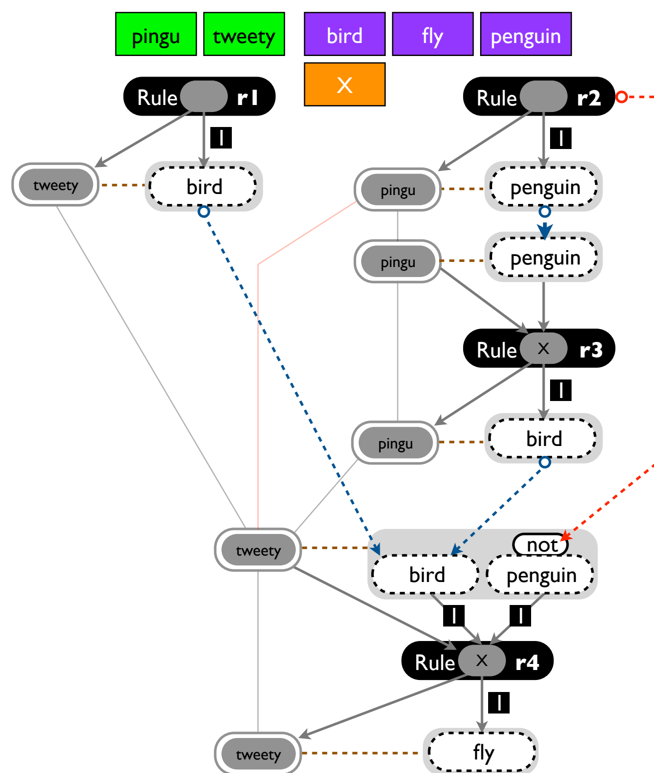


Figure 7.1: The graphical visualization of the *Penguins-Cannot-Fly-Problem*

To allow the construction of an entire ASP program (cf. Figure 7.1), elementary nodes have to be visualized as well. In Figure 7.2 *Variables* (i.e. *X*) are shown as orange boxes, *Predicates* as purple boxes (i.e. *penguin*, *bird*, and *fly*), and *Constants* as green boxes (i.e. *pingu* and *tweety*).

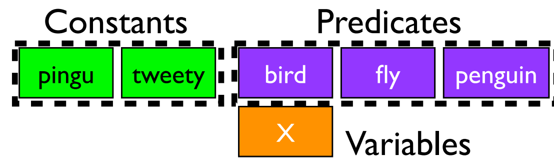


Figure 7.2: The visualization of the elementary nodes of the *Penguins-Cannot-Fly-Problem* program

From the facts shown in Figure 7.3 and Figure 7.4 a set of $E_{SR-Add.}$ edges—visualized as dotted blue edges—or $E_{SR-Subtr.}$ edges—visualized as dotted red edges—can target other nodes.

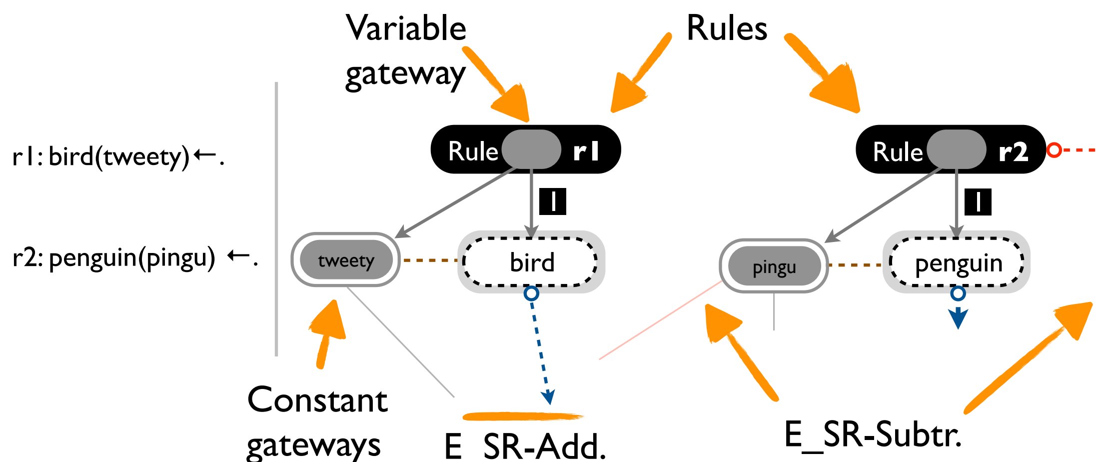


Figure 7.3: The visualization of the facts of the *Penguins-Cannot-Fly-Problem* program

The standard rule *r4* which assures that only *birds* which are no *penguins* can *fly* described in detail in Figure 7.5. This rules inspects the birds *tweety* and *pingu* from rule *r1* and *r3*, and excludes all *penguins* from being processed (*pingu* provided by rule *r2*).

Binary Logic

The second problem is named *3-Coloring-Problem*. It represents a restriction of the *Graph Coloring Problem* discussed in Section 9.2 to three constant colors. This example allows to highlight the variable interactions, the necessity of *ordering labels* and to provide a more realistic usage scenario. Basically, this problem aims for coloring vertices *vtx* in such a manner that neighboring vertices do not hold the same color. This usage scenario requires the following rules:

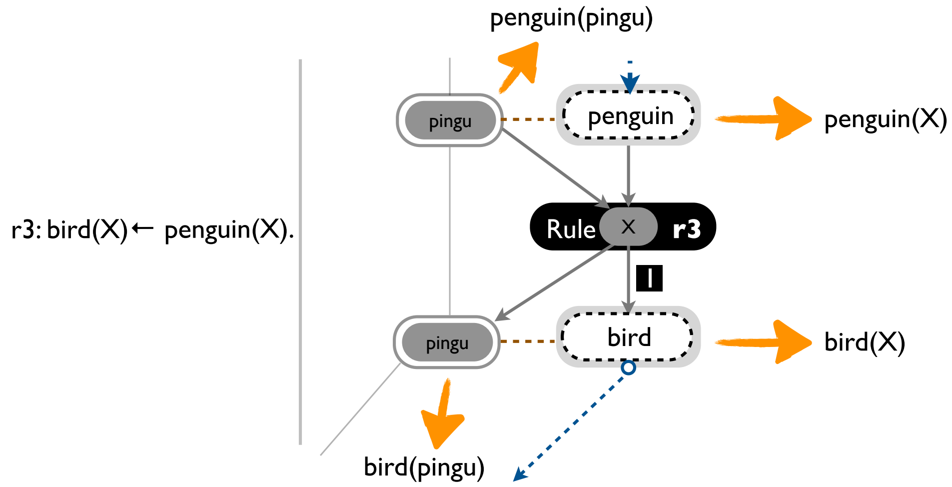


Figure 7.4: The visualization of the facts of the *Penguins-Cannot-Fly-Problem* program

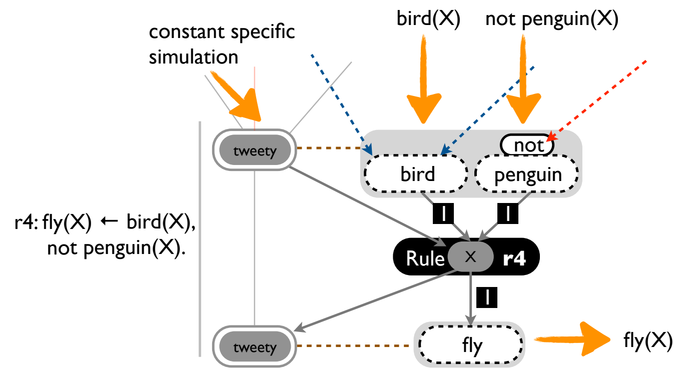


Figure 7.5: The visualization of a standard rule of the *Penguins-Cannot-Fly-Problem* program

$$
\Pi_{7.2} = \begin{cases} r1a : edge(b, a) & \leftarrow . \\ r1b : edge(c, a) & \leftarrow . \\ r2a : vtx(a) & \leftarrow . \\ r2b : vtx(b) & \leftarrow . \\ r2c : vtx(c) & \leftarrow . \\ r3 : chlrd(V, red) & \leftarrow \text{not chlrd(V, green), not chlrd(V, blue), vtx(V).} \\ r4 : chlrd(V, green) & \leftarrow \text{not chlrd(V, red), not chlrd(V, blue), vtx(V).} \\ r5 : chlrd(V, blue) & \leftarrow \text{not chlrd(V, green), not chlrd(V, red), vtx(V).} \\ r6 : & \leftarrow \text{edge(V, U), chlrd(V, C), chlrd(U, C).} \end{cases} \tag{7.2}
$$

One possible solution for the stated problem is shown in Equation 7.2. It represents the underlying basis for the VIDEAS visualization. The rules *r1* and *r2* represent the facts for the constants *a*, *b*, and *c*. The rules *r3* to *r5* are responsible for the initial coloring (*chlrd*) for the vertices (*vtx*). The rule *r6* is the constraint with empty head that removes inconsistent solutions.

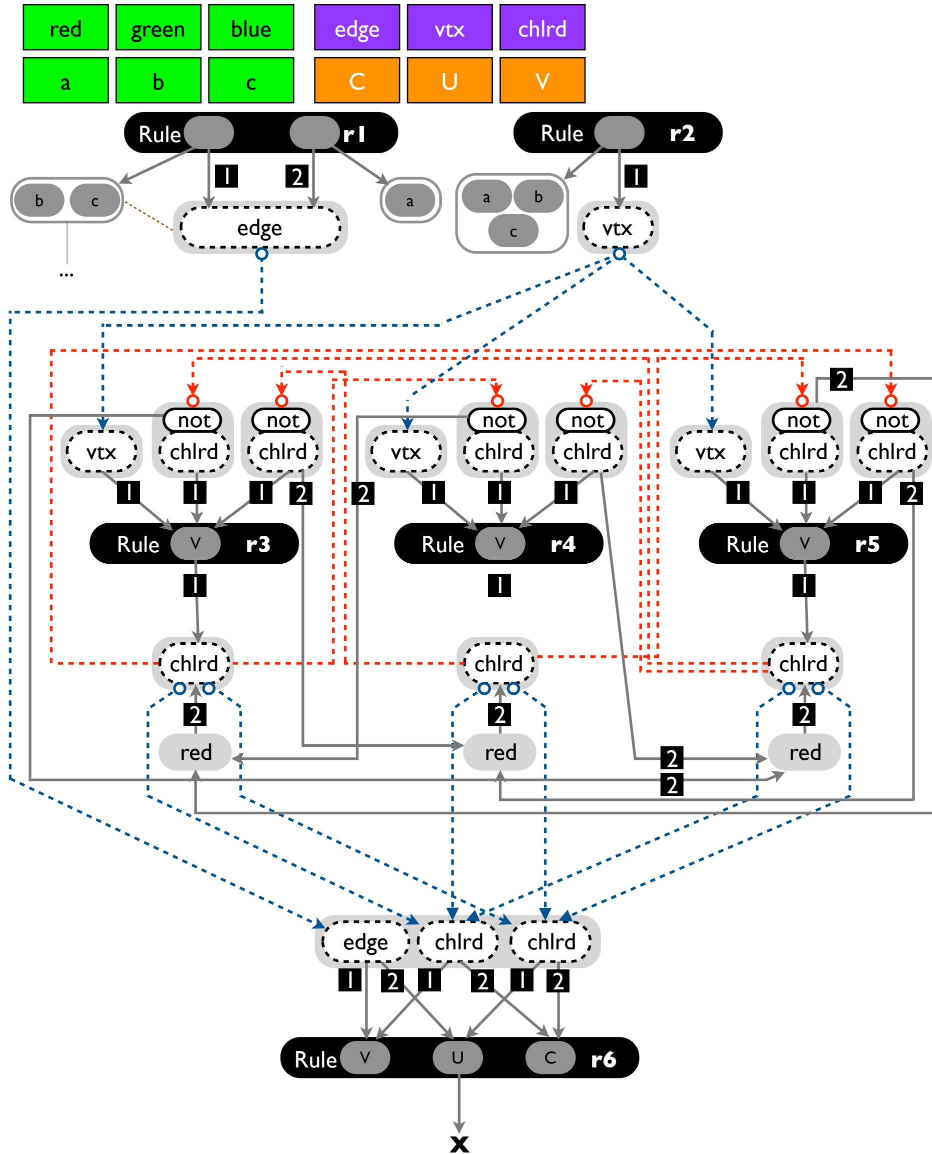


Figure 7.6: A non-disjunctive graphical visualization of the *3-Coloring-Problem*

The details of the program behaviour are explained with the assistance of a VIDEAS model (cf. Figure 7.6). The fact rule *r1* provides two *edges*: *Edge(b,a)* and *Edge(c,a)*. These *edges* are realized by adding a *LiteralGatewayHead* with the expression *edge* which is assigned to two *VariableUsages* as arguments. The constants *b* and *c* are assigned to the first *VariableUsage* re-

ferring to the *Variable U*. The second argument refers to the *Variable v* which is addressed by the constant *a*. These edges represent connections between certain vertices named *vtx*. Therefore, *a*, *b*, and *c* are defined as vertices in rule *r2*. The rules *r3* to *r5* represent the main functionality of this solution. Unfortunately, the required *ConstantUsageBoxes* and the contained *Constant-Gateways* could not be added to the graphical representation of Figure 7.6 as a result of a GMF related technical issue. Adding any further element could not be realized which could reflect a technical boundary of GMF at the moment of writing. In particular, a GMF has only been possible to visualize a maximum number of elements placed on the canvas—other elements have been ignored. Nevertheless, these three rules provide an initial coloring of each *vtx* for which none of the other two colors was assigned. For this purpose, the three constants *red*, *green*, and *blue* are used as the three color values. They are directly used as arguments of the literal *chlrd* which proves or sets the coloring. Each of the three rules sets the color of an uncolored *vtx* to one of these constant colors. As this coloring process from rule *r1* to *r5* only guarantees that all constants satisfying *vtx* are colored in one particular color, it has to be checked in constraint rule *r6* if the colors are the same as for a neighboring *vtx*. Two vertices are neighbors if they are connected by at least one *edge*. For this purpose, the rule *r6* consumes all *edges* bound to the variables *V* and *U*. If the vertices matching to *V* or *U*, respectively, have the same color expressed by variable *C*, the potential answer set is no valid answer set and has to be removed.

This example clearly demonstrates the complexity increase and technical boundaries of this visualization approach. A lot of information is packed on a very small canvas. Several crossings or close placements of connections are necessary to allow the graphical representation in the actual visualization approach. For this purpose, the stated solution for the *3-Coloring-Problem* is modified by using disjunctive rules. This modification reduces the number of rules and their *Edges* (cf. Equation 7.3).

$$\Pi_{7.3} = \begin{cases} r1a : edge(b,a) & \leftarrow . \\ r1b : edge(c,a) & \leftarrow . \\ r2a : vtx(a) & \leftarrow . \\ r2b : vtx(b) & \leftarrow . \\ r2c : vtx(c) & \leftarrow . \\ r3 : chlrd(V,red) \vee chlrd(V,green) \vee chlrd(V,blue) & \\ & \leftarrow vtx(V). \\ r6 : & \leftarrow edge(V, U), chlrd(V, C), \\ & \quad chlrd(U, C). \end{cases} \tag{7.3}$$

The rules *r3* to *r5* of Equation 7.2 are simplified to the single disjunctive rule *r3* which expresses the same behaviour as the original rules. In the visualization (cf. the screenshot of Figure 7.7) the rules *r1a* to *r2c* (cf. Figure 7.9), and the rule *r6* (cf. Figure 7.11) remain identically. The disjunctions of rule *r3* (cf. Figure 7.10) are visualized by using *KnowledgeSpaces* for each *LiteralGateway* for *chlrd*. Each of these disjunctively connected *LiteralGateways* is used to express one constant color, e.g. *red*.

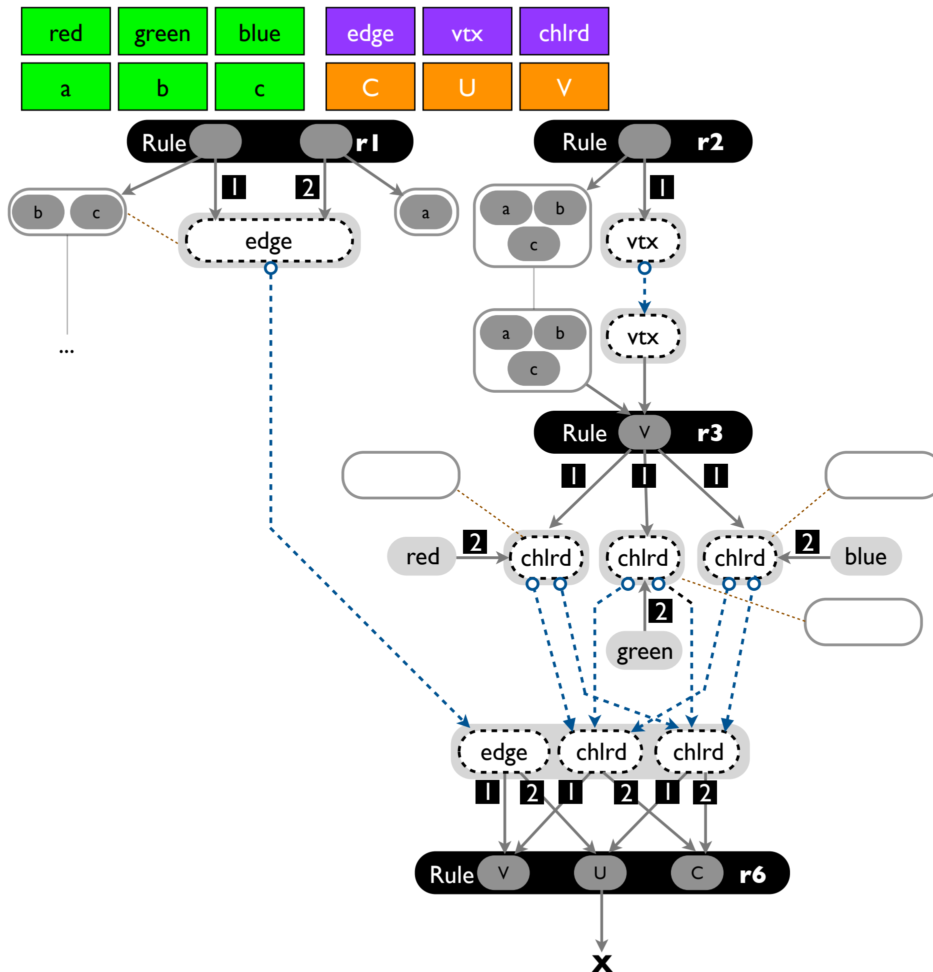


Figure 7.7: A disjunctive graphical visualization of the *3-Coloring-Problem*

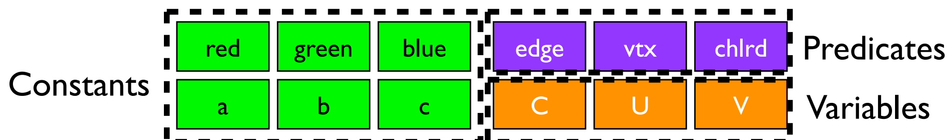


Figure 7.8: Visualization of the elementary nodes of the disjunctive *3-Coloring-Problem* program

The elementary nodes of this visualization for program $\Pi_{7.3}$ are shown in Figure 7.8. In particular, the constants *red*, *green*, *blue*, *a*, *b*, and *c* in the form of green elementary nodes are placed on the canvas. Furthermore, the predicates *edge*, *vtx*, and *chlrd* are shown as purple elementary nodes. The variables of the program—i.e. *V*, *U*, and *C*—are visualized by orange elementary nodes.

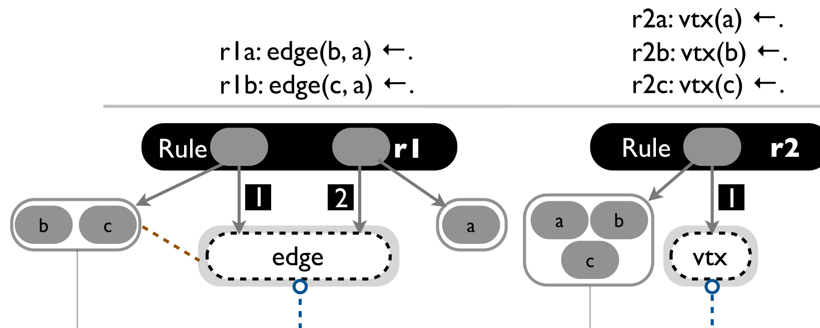


Figure 7.9: Visualization of the fact rules of the disjunctive *3-Coloring-Problem* program

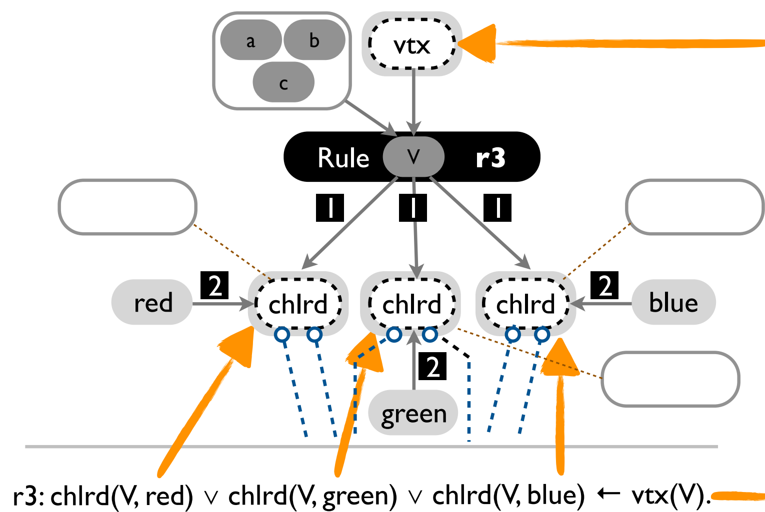


Figure 7.10: Disjunctive rule for a visualization of the *3-Coloring-Problem*

The five fact rules of program $\Pi_{7.3}$ are visually aggregated to the two comprehensive rules *1* and *2* (cf. Figure 7.9). This is achieved by referencing the constants *a*, *b*, and *c* to two constant-independent rules. A disjunctive standard rule is shown in detail in Figure 7.11. In this visualization three *knowledge spaces* are used holding one head literal each, e.g., *chlrd(V, red)*. A special case is comprised in the constraint rule *r6* which uses the predicate *chlrd* as two literals (cf. Figure 7.11). These literals check if two different vertices matching variables *V* and *U* have the same color—this is ensured with variable *C*. Additionally, it is checked if an *edge* from constants matching *V* to constants matching *U* exists.

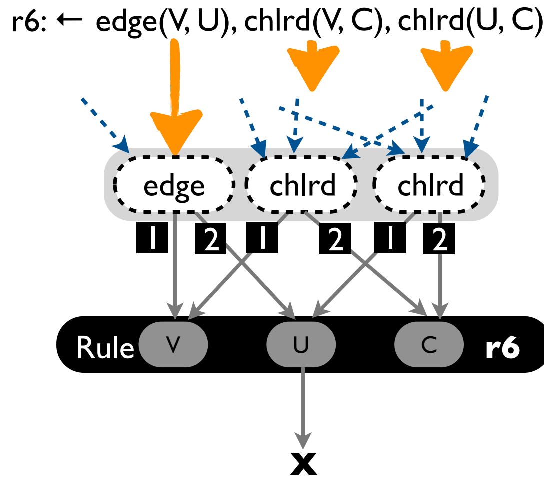


Figure 7.11: Constraint rule for the visualization of the disjunctive *3-Coloring-Problem* program

7.2 Visualization

The visualization capabilities of VIDEAS are evaluated in three categories: (1) The complexity, (2) the quality, and (3) its ability to support analysis. Each of these categories is evaluated in respect to the stated usage scenarios.

Complexity

The complexity of a graphical VIDEAS model relates to the number of used rules and their interrelations. The number of added *Constants*, however, only has a minor influence on the visualization complexity. Of high importance for the involved complexity are the defined *LiteralGateways* of the rules. The more *LiteralGateways* a rule uses the more likely is an interaction with other rules. Therefore, *LiteralGateways* have to be connected with a rule and, furthermore, can share or receive information from other rules.

For the *Penguins-Cannot-Fly-Problem* shown in Figure 7.1 44 elements are necessary to express the underlying answer set program. These 44 elements contain three rules, five *Literals*, and two *Constants*. For the *3-Coloring-Problem* of Figure 7.6 more than 144 elements are necessary from which 6 are rules, 17 are *Literals*, and three are *Constants*. The 144 elements represent the actually shown elements. However, the *Constants* of the rules *r3* to *r5* could not be added to the diagram (resulting from the GMF implementation boundaries). The simplified variant shown in Figure 7.7 reduces the effort to 97 elements including all *ConstantUsageBoxes*, but not involving an arbitrary combination of *ConstantGateways* in a state. The adding of a further graphical element increases the complexity by a specified value range. Such a range is for example [2, 2+SR_Edges_{in}/SR_Edges_{out}] for a *LiteralGateway*. This range is mainly influenced by the number of *Edges* sharing information with other rules. The more rule dependencies exist, the more hidden information can be visualized and, therefore, requires more graphical elements.

$$
\begin{aligned}
overall() :& \\
& elementaryNodes() + rules() \\
elementaryNodes() :& \\
& |Literals| + |Constants| + |Variables| \\
literalGtws(\text{rule r}) :& \\
& |LiteralGateways| + |\mathit{KnowledgeSpaces}| + |D_Edges| + \\
& + |SR_Edges| + |SR_AS_Edges|) \\
\mathit{constantUB}(\text{rule r}) :& \\
& |\mathit{ConstantUsageBoxes}| + |ConstantGateways| + \\
& + |D_Cons_Edges| + |SR_Cons_Edges| + |KS_Edges| \\
rules() :& \\
& R + \sum_{r=1}^{R}(|VariableUsage| + (|\mathit{ConstraintBoxes}| + \\
& + |\mathit{Constraints}|) + \mathit{literalGtws}(this) + \\
& + \mathit{constantUB}(this))
\end{aligned}
\tag{7.4}
$$

A more general description of the VIDEAS complexity is shown in Equation 7.4 as pseudo code returning the number of required elements on the canvas. The function *overall* provides the complexity of Π based on two functions *elementaryNodes()* and *rules()* (for *R* rules contained in Π). The function *elementaryNodes()* establishes the elementary nodes reused by other elements: *Literals*, *Constants*, and *Variables*. The function *rules()* represents the rule-centric elements of Π. These elements are the rule itself, the contained *VariableUsages*, the *LiteralGateways* (including the *KnowledgeSpaces* they may depend to), the *ConstantUsageBoxes* and their containments, as well as optionally a *ConstraintBox* with a number of *Constraints*. The complexity calculations of *LiteralGateways* and *ConstantUsageBoxes* are placed in the functions *literalGtws()* and *constantUB()* respectively.

This pseudo code is used to demonstrate that the definition of rules is critical for the complexity of Π. Furthermore, it supports the analysis from the two exemplary usage scenarios of this section by highlighting the complexity of *LiteralGateways* in comparison to *ConstantGateways*. As a consequence, the adding of new rules is critical for the complexity of the visual representation of Π as it involves many related elements. *LiteralGateways* with their number of direct relationships are the critical elements of a rule. Therefore, both elements are disproportionately responsible for complexity increases of the overall graph and, therefore, have to be reduced, if possible. For this purpose, only a beneficial subset of the number of rules of Π should be visualizable as well.

Although in this section additional effort for visualization is discussed, the provided perspective on ASP programs may be a valuable asset in designing and analyzing programs. The visualization allows the developer to set the focus on design issues rather than on implementation details. In particular, this allows the concentration on program design decision only, eliminat-

ing the effort for program code specific decisions in the software development. Moreover, this perspective may only be applied by developers, whenever classical ASP software development techniques have proven to be inefficient.

Quality & Environmental Boundaries

The graphical model provides a single artifact visualizing the behaviour of Π. In the code this behaviour is spread over a set of rules. Their interrelations are not directly specified in the code, but are clearly visible in the graphical model (cf. Figure 7.1 and 7.6). This graphical model is, therefore, able to highlight which *LiteralGateways* are potentially violating a constraint rule, is sharing information with other rules, or is preventing the execution of successor rules. This new perspective on Π is the most essential asset provided by VIDEAS. Moreover, the design-specific issues are decoupled from their code by making use of an abstract visualization which allows the focussing on essential elements and, therefore, advocates the focussing on essential decisions regarding the behaviour of Π. The complexity of understanding a solver-specific syntax is removed and, therefore, allows a design process which is independent of particular textual notations. This is a further asset for the interoperability of Π.

Each used element is clearly labeled or can be recognized by comparing shapes and colors, e.g., the literal gateway *bird* is labeled with “bird”. Furthermore, the rules as most essential elements are visually set apart of all elements. This is a great asset in recognizing the behaviour of Π. However, the visualization could be optimized such that semantical dissimilar elements are visually separated even more effectively. Nevertheless, the produced graphical model can be used for communicating the behaviour of Π and documenting it for further extensions or maintenance work. In the professional software engineering sector the communicability and ability to document a program, is already valued as highly important factor. So, it is essential for answer set programs as well to support such features conveniently which can be addressed by applying the VIDEAS approach.

The visualization is, however, hampered by the disproportionally growing visualization complexity which challenges actual visualization techniques and the human capability of recognizing the essentials on the graph. In particular, problematic is the number of crossings and closely placed elements. Such element placements increase the effort in recognizing interrelations of elements described by *Connectives*. Furthermore, only a certain number of elements of a defined size can be placed on the canvas in such way that they can be recognized at once. The technical boundary of GMF’s capability of managing a greater number of graphical elements is an additional drawback. It is, therefore, necessary to hide some elements placed on the canvas. Another technical boundary of GMF is that the addition of elements sometimes is hindered by some GMF bugs. Nevertheless, the capabilities provided by GMF allow the easy definition of own graphical models such as VIDEAS. As GMF is a dynamically developed Eclipse project, some of the stated issues are probably removed in future versions for which VIDEAS can be adopted.

Another boundary of GMF is the problem of inconsistent visualizations. In particular, elements based on the same graphical definition are sometimes visualized differently in different contexts. An element used as *Compartment*, e.g., *LiteralGateways*, is sometimes differently visualized than as *Top Node Link*, although their visualization is defined identically. This hand-

icap, however, does not strongly interfere with the ability to evaluate VIDEAS as done in this section.

Analysis

For the analysis of the program Π three areas are inspected: (i) Recognizability of Π's essentials, (ii) the application of constraint rules, (iii) the recognition of similar patterns, and (iv) the application of *not*-negations.

(i) The essential elements of Π are rules. These rules are easily recognizable in VIDEAS (cf. Figure 7.6). Especially the dependencies of rules—from which rules they consume and which other rules consume from them—are visualized in VIDEAS. These dependencies cannot be directly shown in the textual representation and are, therefore, a very valuable additional information for recognizing problematic or even erroneous elements of Π. Common pitfalls in the design of answer set programs are often related to misspecification of such rule dependencies. Such pitfalls can be identified by interpreting the *Edges* pointing to or from a rule to other elements (cf. Figure 7.12).

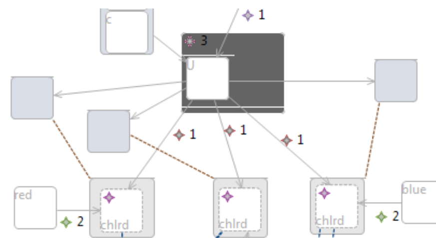


Figure 7.12: Visual rule-centricity leading to a multitude of elements associated to rules

(ii) Constraint rules (cf. rule *r6* Figure 7.6) consume inputs via special *Edges*. These *Edges* demonstrate dependencies of head literals with constraint rules and, therefore, help to identify why a constraint rule is violated for a particular set of data. This is a valuable asset, but optimally such constraint rule applications could automatically highlight for which path a violation exist. A highlighting of a violation has not been implemented yet. This, therefore, represents an intended enhancement for the future.

(iii) The example of Figure 7.6 demonstrates that equivalences and similar patterns of rules are highlighted by the usage of visual representations. The rules *r3* to *r5* are structurally similar representing different cases. To optimize the capability of recognizing similar elements algorithms could help to identify similar patterns. Furthermore, the disabling of uninvolved elements could allow better communication of existing patterns.

(iv) The *not*-negation is represented in the graphical representation as subtraction of *LiteralGateways* and *ConstantGateways*. The subtraction of *ConstantGateways* from a set can be identified in Figure 7.1 where in rule *r0* all *penguins* are subtracted from the set of shared *ConstantGateways* as *birds*. This allows a good overview of actual states of Π and, therefore, supports the understanding of Π's behaviour. Nevertheless, this step would be optimally supported by automated constant forwarding mechanisms.

7.3 Generated Code

The VIDEAS visualization of Figure 7.1 expressing the *Penguins-Cannot-Fly-Problem* can be transferred back to a textual answer set program representation by running the code generator *UnaryTemplate* producing code runnable in DLV (cf. Equation 7.5). Executing this code returns a single answer set: *{bird(tweety), bird(pingu), penguin(pingu), fly(tweety)}*. The good performance and the high output quality allow a seamless switching of Π's representation which is a key-factor for the applicability of VIDEAS.

$$\Pi_{7.5} = \begin{cases} r1 : bird(tweety) & :- . \\ r2 : penguin(pingu) & :- . \\ r3 : bird(X) & :- penguin(X). \\ r4 : fly(X) & :- \textit{bird(X), not penguin(X).} \end{cases} \tag{7.5}$$

The more complex example of Figure 7.7 can be transformed to answer set program code as well. For this purpose, the n-ary code generator of the file *NaryTemplate* is used. This file is not able to produce all language constructs. The result is shown in Equation 7.6 in an unmodified DLV syntax.

$$\Pi_{7.6} = \begin{cases} r1a : edge(b, a) & :- . \\ r1b : edge(c, a) & :- . \\ r2a : vtx(a) & :- . \\ r2b : vtx(b) & :- . \\ r2c : vtx(c) & :- . \\ r3 : chlrd(V, red) \text{ v } chlrd(V, green) \text{ v} & \\ \quad \text{v } chlrd(V, blue) & :- \textit{vtx(V).} \\ r6 : & :- \textit{edge(V, U), chlrd(V, C),} \\ & \quad \textit{chlrd(U, C).} \end{cases} \tag{7.6}$$

In this equation, the facts are not generated which entails a necessity for code modifications to allow an execution on solvers such as DLV. For this reason, the generation of facts represent a required future enhancement of the proposed generator. However, both code generators strongly increase the suitability and applicability for professional/academic usages or experiments. So, they represent a useful advancement for VIDEAS and answer set programming in general.

7.4 Summary

The VIDEAS approach provides a new perspective on answer set programs, which allows the focussing on the essentials. The recognition of misspecified rule dependencies or other common

errors is better recognizable with VIDEAS. Even common patterns can be easier identified and communicated. The solid code generation provides a good transition to executable answer set programs. However, the visual representation and especially the technical environment shall be optimized in future versions. The code generation requires further optimizations in terms of support for higher arities, and is optimally complemented by a counterpart parsing answer set program code to visualize VIDEAS models.

CHAPTER 8

Lessons Learned: Usability Enhancement

According the high visualization complexity of VIDEAS some usability enhancements are necessary. Possible enhancement variants are briefly discussed in this section. This resulting approach is based on VIDEAS and is therefore named VIDEAS$^+$. In lieu of applying these proposed usability enhancements of VIDEAS$^+$, alternative problem- or environment-specific optimizations could be valuable enhancements of VIDEAS$^+$. However, in this thesis a general usability enhancement is introduced.

8.1 Optimization Strategies

In particular, the following three main optimizations for the VIDEAS visualization are proposed in this section to reduce the visualization complexity: (i) Redundancy minimization, (ii) decoupling of solver-specific capabilities, and (iii) hiding of implicitly expressed information.

(i) In VIDEAS, the redundancy of elements mainly expressed by the usage of gateway nodes represents an asset in recognizing different sort of dependencies organized around ASP rules. However, these redundancy non-linearly increases the visualization complexity while adding new program statements. For this purpose, this redundancy is immolated by moving to a resource-based perspective of ASP programs. Such resources are mainly predicates which are used as literals in ASP rules. Each resource can be used *1* to *n* times, but is only placed once on the canvas. For this purpose the usage modalities, e.g., negations, different cases or equalities, have to be handled by transitions dedicated to a rule.

(ii) Another complexity issue of VIDEAS can be related to its intention of providing visualizations closely related to the original program representation. On the one hand this is a great asset as the dependencies of rules in the visualization are directly related to textual rules. On the other hand visualizations integrating the capabilities of solvers or technical restrictions limit the visualization power, if the relationship between ASP code or ASP solver and the visualization is

too close. In particular, the usage of inheritance patterns and connectives might not be supported by solvers, but allow a more compact representation in the visualization. Therefore the usage of visualization-specific elements aggregating code elements is enforced in VIDEAS+.

(iii) Not all elements in a VIDEAS visualization are necessary for understanding the behavior of an ASP program. Especially, the heavy usage of labels can be reduced by introducing conventions. In VIDEAS+, the top-to-bottom and left-to-right reading of elements is introduced as convention. This implies that the first element from the top-left of a graphical element, e.g., literal gateway, corresponds to the first, the very right element at the bottom to the last argument of a textual code block, e.g., literal. Consequently, such a convention can also be used for the naming of element such as variables. As each variable gateway is handled as usage of a unique variable, the naming is irrelevant at the design-time of ASP programs. For this reason the variable names can be randomly added when generating the ASP code. Each element, which is not addressed by head and body literals, is unbound.

8.2 Visualization of Transitions

One crucial factor to realize these optimizations is the efficiency of the visualization of such transitions. There are several possible options which are pointed out in Figure 8.1. Transitions could be established by adding directed relationships—cf. (a) in Figure 8.1—from one predicate to another or even one predicate attribute to another (based on the syntax of ER-diagrams). However, such transitions are not applicable for expressing cases based on attribute conditions, e.g., inequalities, and cannot be sufficiently visualize which relationships (edges) are dependent on each other. Therefore, this visualization variant cannot express powerful ASP programs.

A puristic transition alternative based on a self-defined syntax is given in (b) of Figure 8.1. In this variant, each line of a transition represents a case. These lines provide a set of term gateways which can be shared be several cases or be used independently, e.g., *chlrd(X, C)* and *chlrd(Y,C)* where *chlrd(. . .)* are cases. The variables *X* and *Y* (represented as gateways) are dedicated to one of these cases, and *C* (vertical box on the right of the transition) is a variable which is used in the visualization of both literals. To reduce the number of edges a vertical term gateway is introduced, which allows the specification of conditions with rule-wide scope, i.e., *X* and *Y* are both elements of *vtx* independently of the case the depend to. These transitions semantically represent cross-products of all instances—comparable to matching behavior of graph transformations [44]—satisfying all conditions specified by this transition. As this implies permutations of all instances consumed by each term gateway, are produced as result by this transition, the semantics of such transitions is difficult to understand and hampers the ease of program development.

A more comprehensible transition approach is provided by Colored Petrinets (CPN) [24] which can clearly express value mappings by color patterns—cf. (c) in Figure 8.1. Each transition is fired (the rule is applied) whenever all body literals—represented by colored MetaTokens (comparable to nodes)—pointing to this transition are satisfied. The transition then produces a head literal—represented as colored MetaToken—to which an edge points to. To ensure that only corresponding resources are consumed or produced, two-colored MetaTokens are necessary ensuring this relationship within a Transition—this elements are directly target by predicates (a

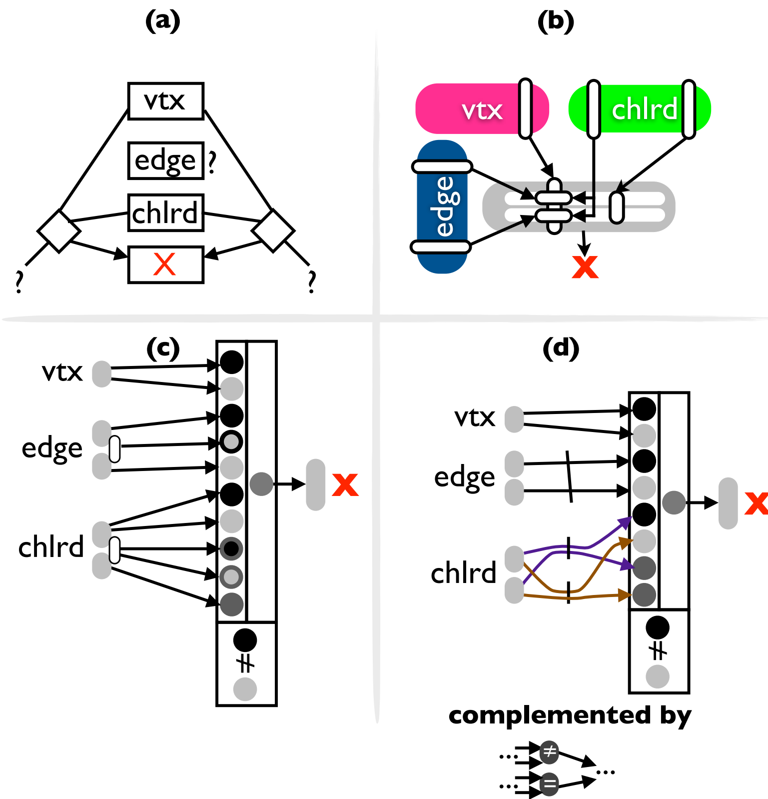


Figure 8.1: Several transition variants as usability enhancements for VIDEAS⁺

box is used representing complete instances of a predicate instead of instances of arguments). Such two-colored MetaTokens hold the color of the literal—border color—and the argument color—inner color. To reduce the MetaToken count, one-colored MetaTokens representing these literals themselves which would be typically used by CPN were eliminated. Consequently, one argument—e.g., the first—is set as mastering argument to which all others refer. The resulting transition behavior is simple and expressive, but still requires a multitude of two-colored MetaTokens which strongly raises the visualization complexity.

For this purpose, a context-specific CPN variant is proposed which tries to establish the bindings of literal arguments (cases) without the usage of two-colored MetaTokens—cf. (d) in Figure 8.1. These bindings are realized by a visual band (cf. the relationships of *edges* are bound in Figure 8.1) binding two consumptions (head literals) together. Furthermore, context-specific primitive relationships such as inequalities or equalities can replace these sophisticated CPN transitions for simple rule constellations. As this variant allows the visualization of sophisticated ASP language features and offers primitive relationships to reduce the visualization effort, this variant is chosen for VIDEAS⁺.

8.3 Example

These three optimizations are shown in the example of Figure 8.2 which is based on the ASP program $\Pi_{8.1}$ which was visualized by VIDEAS in Figure 7.7. Arguments of predicates and MetaTokens of transitions are unlabeled according the new implicit naming convention. The usage of connectives—e.g., *chlrd(..., red)* ∨ *chlrd(..., green)* ∨ *chlrd(..., blue)*, are enforced. The resource-centricity is most obvious for the predicate *chlrd* where values are provided by one transition (more different transitions could provide values as well) and consumed by other transitions—independently of the applied values. The bound edges pointing from *chlrd* to the constraint transition show the visualization of several literals of the same predicate—a case—being consumed by a single transition. Edges being dependent on each other can have the same color to allow an easier recognition, e.g., purple and brown in Figure 8.2. Equalities are enforced by using the same MetaToken color which mean that the same value is consumed several times. The usage of different colors states that different instances are used—to ensure their inequality additional constraints can be specified for each transition.

$$\Pi_{8.1} = \begin{cases} r1a : edge(b,a) & \leftarrow . \\ r1b : edge(c,a) & \leftarrow . \\ r2a : vtx(a) & \leftarrow . \\ r2b : vtx(b) & \leftarrow . \\ r2c : vtx(c) & \leftarrow . \\ r3 : chlrd(V,red) \vee chlrd(V,green) \vee & \\ \quad \vee\, chlrd(V,blue) & \leftarrow \text{vtx(V)}. \\ r6 : & \leftarrow \text{edge(V, U), chlrd(V, C),} \\ & \quad \text{chlrd(U, C).} \end{cases} \tag{8.1}$$

It seems notable that the enforcement of language features simplifying the visualization, i.e., connectives, has to be ensured by supporting techniques in the realization informing the developer of possible simplifications.

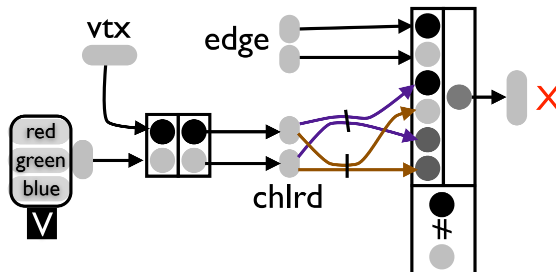


Figure 8.2: The 3-Coloring-Problem visualized by VIDEAS+

As these transitions are not fully trimmed to the textually realization of rules, the generated rules from the graphical model can differ according the capabilities of the solver the code is optimized for. For this purpose, an assistance is necessary for superimposing *rule patterns*. In Figure 8.3 such a rule pattern is provided for a non-disjunctive solver which therefore divides the transition modeling a disjunction—*red*, *green*, and *blue*—into three non-disjunctive rules. The constraint transition can be expressed by a single non-disjunctive rule. Resources involved in transitions are only added to rule patterns, if only a certain configuration, e.g., *blue*, is part of a particular rule.

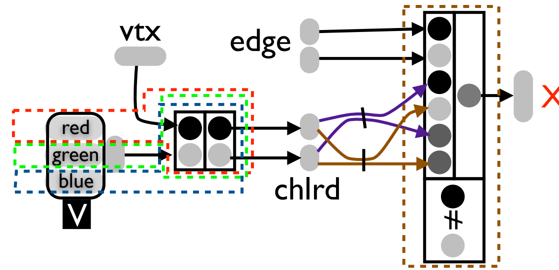


Figure 8.3: Visualizing rule patterns for a non-disjunctive solver with VIDEAS+

CHAPTER 9

Conclusion

9.1 Summary

Over the last years answer set programming (ASP) has been responsible for a great impetus of logic programming. However, the lack of interest in logic programming for professional applications could not be tackled by ASP so far. This might be explained by the absence of a sufficiently supported software development methodology.

For the purpose of counteracting the absence of such methodologies, innovative visualization approaches supporting the conception, analysis, and debugging of ASP programs were proposed in this thesis. First, a concept for non-deductive ASP programs based on ER-diagrams was presented, which is capable of designing structural relationships and furthermore allows the generation of facts and constraints for these facts. As this approach is not capable of visualizing program structures necessary to understand the program behavior and is not capable of handling deductions, a more complex method is necessary. In the second step, these drawbacks were therefore addressed by proposing an approach which allows the stepwise visualization of deductions comprised in ASP programs. The approach was named VIDEAS and was realized in a prototypical implementation with the usage of Eclipse EMF, GMF and M2T projects. This realization includes a graphical diagram editor and a code generator. However, in the evaluation of VIDEAS its complexity challenging modern visualization techniques highlighted the necessity for further advancements. As a consequence, in a final step usability enhancements were undertaken reducing the visualization effort of VIDEAS—this approach named VIDEAS^{+} provides some ideas of possible enhancement scenarios and underline the applicability of visualization approaches for ASP development purposes. Additionally, an outlook on advanced features such as cardinalities, weights, and connectives is given in the following section.

9.2 Outlook on Advanced Features

Beyond the focus of visualizing basic features of deductive ASP programs Π (cf. Section 5), some advanced language features can be integrated in the visualization approach as well. This section gives a brief outlook on the introduction of such features. In particular (i) connectives, (ii) cardinalities and weights, and (iii) inheritance patterns are inspected. Some of the listed features are already integrated in the VIDEAS implementation.

Connectives

The connectives in ASP have an implicit and explicit nature. In this section, therefore, connectives are considered from the perspective of implicit and explicit usages, as well as inter-compatibility of connectives.

Connectives are explicit whenever a keyword or a graphical counterpart is required to express a connective relationship between two elements, e.g., in *a or b* the disjunction (*or*) represents an explicit connective. Conjunctions ($\wedge/AND$) and disjunctions ($\vee/OR$) mainly allow the differentiation, if literals can or cannot be applied within the same candidate answer set. Recall, that each body literal is implicitly in a conjunctive relationship (only separated by commas), whereas head literals are automatically in a disjunctive relationship (cf. Figure 9.1). This implies that all body literals have to be satisfied to apply the rule comprising them.

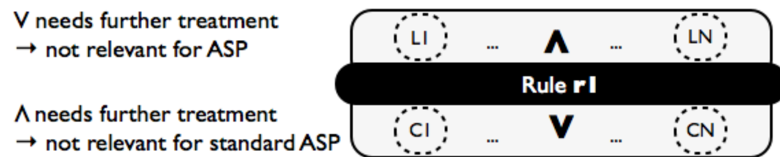


Figure 9.1: The semantics of the implicit connectives in ASP rules

In ASP, the enforced disjunctions between head literals are explicitly noted. In addition to these commonly used explicit disjunctions in ASP programs, explicit conjunctions could textually and visually extend the ASP concept as well. Explicit conjunctions are of high relevance for other logics such as predicate logic, and therefore could find their way in ASP as well. Generally, ASP programs have a very restricted structure concerning the usage of connectives. The VIDEAS approach provides the flexibility to model formulas of arbitrary structure in a holistic manner. As a consequence a holistic visualization concept for unenforced, enforced, implicit and explicit connectives is given in the following sections.

Visualization

Implicit or explicit connectives can be visualized by using *knowledge spaces* (new metamodel class: *KnowledgeSpaceBody* (body literals) or *KnowledgeSpaceHead* (head literals)). Graphically all literals being placed in the same *knowledge space* are accessible at the same time by successor rules. If no *knowledge space* is added to the graph, the implicit connectives—as in-

troduced above—are used. Nevertheless, it is highly recommended to make use of *knowledge spaces* to ensure the consistency of the visualization.

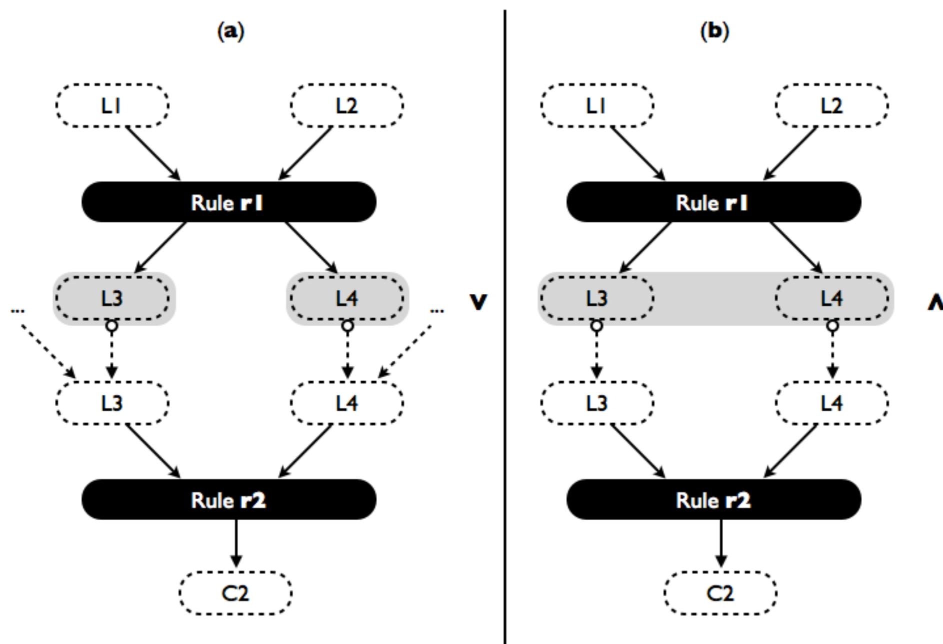


Figure 9.2: The accessible knowledge of ∨ and ∧ connected literals

Such *knowledge spaces* are visualized with the head literals $L3$ and $L4$ in Figure 9.2. On the left side (a) two ∨ connected head literals are visualized. Both of these head literals are contained in own *knowledge spaces*, which are visualized by a grey background for all connected literals in the same space—in this case only one literal for each space. The same constellation with ∧ connected head literals is shown in (b) of Figure 9.2. The literals $L3$ and $L4$ share the same *knowledge spaces* and are, therefore, both accessible for following rules. As both literals—$L3$ and $L4$—are required to be satisfied to apply rule $r2$, (a) requires different sources for $L3$ and/or $L4$ to be somehow applied. To apply it for both constellations ($L3$ and $L4$ being consumed from rule $r1$) both literals need additional sources. In the simplest case this is again a rule returning $L3 \vee L4$. Further sources for $L3$ or $L4$ are not required in example (b) as both can be consumed from rule $r1$. At this point it is again necessary to highlight that the constellation of example (b) in Figure 9.2 does not show a syntactically correct ASP program, i.e., ∧ connectives are unsupported.

Inter-Compatibility & Hierarchization. The introduction of connectives creates the opportunity of nested connective usages producing extremely sophisticated formulas. As such formulas, however, mostly do not directly correspond to ASP programs, they are only briefly sketched in this section. Some inter-compatibility scenarios are introduced in Figure 9.3. For known solvers only variants (a) to (c) are applicable. Variants (d) to (f) are nearby extensions that highlight the extendability of the presented approach.

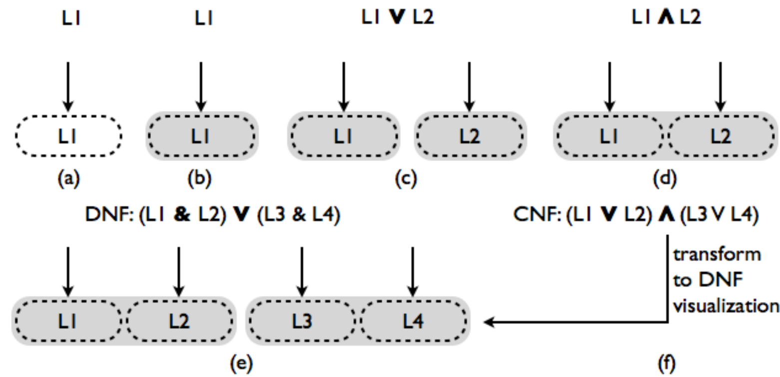


Figure 9.3: The variations of ∨ and ∧ within an ASP rule

The variants (a) and (b) show semantically identical non-disjunctive literals—(b) uses an additional *knowledge space* to ensure the visualization consistency. The variant (c) introduces disjunctive literals with the example $L1 \vee L2$. Both literals of (c) are in different $knowledge$ $spaces$ and, therefore, cannot be used in the same successor rule at the same time. Conjunctive literals are introduced in variant (d) where $L1$ and $L2$ share the same *knowledge space*.

These atomic connections can be combined to formulas which are then transferrable to disjunctive normal form (DNF) or conjunctive normal form (CNF). DNF encapsulates conjunctive literals in disjunctions, e.g., $(L1 \wedge L2) \vee (L3 \wedge l4)$ (cf. (e) in Figure 9.3). Each ∧ connected block (marked by the brackets) shares the same *knowledge space*—limiting the accessibility of literals. As the blocks are ∨ connected, they do not share their space. In contrast the visualization of CNF—(f) in Figure 9.3—is only possible by introducing a nesting of knowledge spaces which hampers the ease of understanding. Consequently, a transformation to DNF is enforced.

Cardinalities & Weights

For a set of body literals of a rule, cardinalities (new metamodel class: *Cardinality*; cf. Figure 6.1) define a threshold of literals to be satisfied—the minimum and maximum number of body literals, which have to be contained in the answer set—in order to apply the rule. Such constraints are supported by solver extensions, e.g., for SMODELS [37].

The textual cardinality is expressed by a lower and an upper bound. Graphically cardinalities have to be recognizable at a glance as well. For this purpose, they are directly visualized as labels of *knowledge spaces* as shown in Figure 9.4—(b) to (d) highlight lower and upper bounds (e.g., $[2, 3]$ represents the lower bound *2* and the upper bound *3*). Weights (new metamodel class: *Weights*) in contrast provide a decimal weight (class: *WeightValue*) for each body literal. Whenever the sum of the weights of the body literals are greater than the lower bound or equal (as decimal) and less or equal than the upper bound, the rule can be applied—cf. (e) to (h) of Figure 9.4.

Another usage variant is offered by rule specific cardinalities and weights (cf. (a) and (e) of Figure 9.4) which are not yet supported by solvers. A nesting of cardinality/weight blocks (including rule specific cardinalities/weights) can allow a precise “factor weighting” of literals.

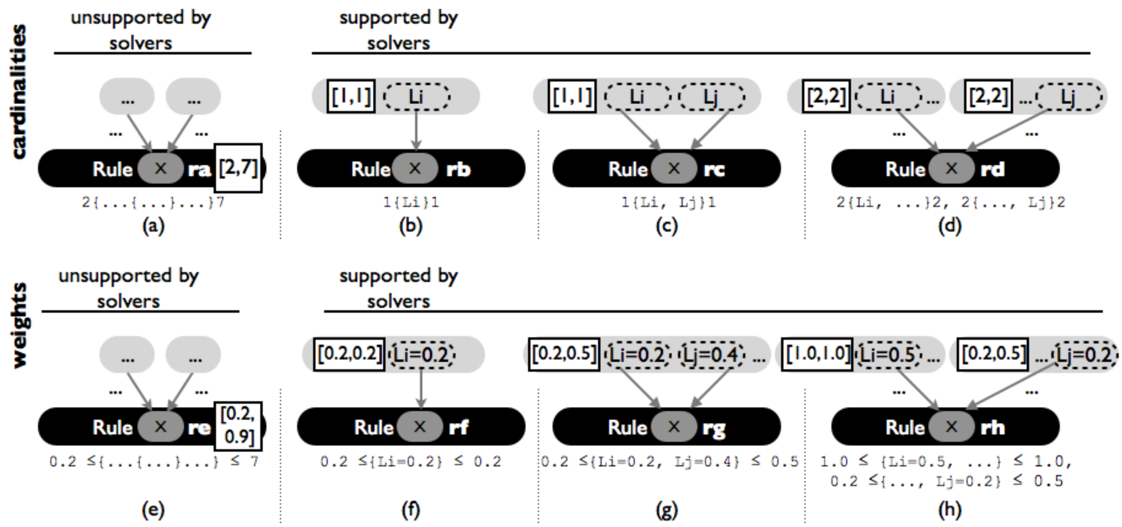


Figure 9.4: Graphical cardinalities (a-d) and weights (e-h) for ASP programs

Two further modifications are necessary to visually support the usage of *knowledge spaces* for *cardinalities* and *weights*. First, more than one *constant usage box* for consumed and produced constants have to be useable for each *variable usage*. Second, each *constant usage box* refers to (at most) one *knowledge space*.

Extraction of Inheritance Patterns

Often answer set rules describe generally applicable patterns. In this section, one particular pattern—the inheritance relationships describing inheritances of literals, e.g., all *Cars* are *Vehicles* as well—is extracted from answer set rules. Inheritances are implicitly expressed in ASP programs by using additional rules. To simplify such relationships in the visualization, inheritances should be shown decoupled from other rules (cf. the relationship on the right of Figure 9.5). As a consequence aggregating edges (marked with *a* in Figure 9.5) can be used pointing from subliterals to superliterals. The resulting deductions are more directly applicable in comparison to the previously introduced visualization (cf. on the left of Figure 9.5).

Technical Realization

Additional effort is required in the technical realization of VIDEAS in order to support the presented advanced features. In particular, a set of further classes according the chosen advanced features have to be added to the meta model (cf. Section 6) and have to be complemented by a visual representation used in the graphical editor. Consequently, the code generators have to be enhanced as well. In this thesis, the advanced features of cardinalities, weights, and connectives are technically integrated in VIDEAS.

The graphical editor provides the capabilities to visualize all these advanced features. Even the *UnaryTemplate* code generator could be adopted to match these proposed advancements. If

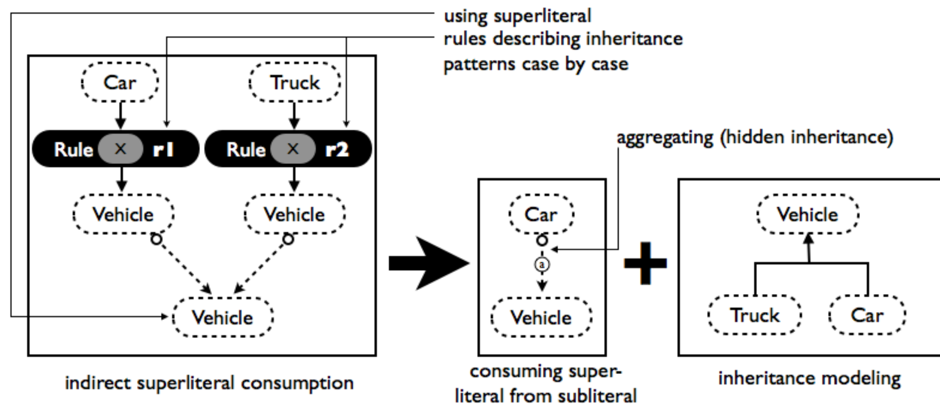


Figure 9.5: Extracting general guilty patterns (inheritance) from rules

required by a solver, the code generator can ignore unsupported language concepts, e.g., cardinalities and weights for DLV. The standard settings of the *UnaryTemplate* focuses on the capabilities of DLV. Another possible optimization would be the generation of an intermediary format program code (supporting all advanced language constructs), which is in a post-processing step adopted to the specific capabilities of the used solver. This may be beneficial in order to allow custom post-processors for a variety of solvers.

9.3 Future Work

This section is intended to introduce ideas for future extensions and approaches beyond the boundaries of this thesis. First of all, it is highly important to add the actual ASP program execution state in the visualization. This can only be achieved by integrating a solid solver, e.g., DLV. Such a solver may help to highlight which constraint *Rule* is violated by which *ConstantGateway* on which path. It, moreover, allows the stepwise proceeding of the execution and therefore provides intermediary states. In addition, the establishment of round-trips from a textual answer set program Π via a graphical model back to a modified textual representation of Π is another important required capability. At the moment the initial process from Π to its graphical model is not automated. This, however, is essential to support the optimal integration of VIDEAS in the ASP development and to allow a dynamic switching between representations.

Not only the generation of an initial graphical model should be automatable, but every step which involves the forwarding of elements. In particular, *ContantGateways* are additively and subtractively shared with a set of successor *Rules*. In the current prototype these *ConstantGateways* have to be added to every *ConstantUsageBox* by hand although the abstract representation already enables a forwarding. Furthermore, the identification of successor rules for *LiteralGateways* can be automated.

To support the quality of ASP programs, used patterns should be highlighted. Furthermore, some standard patterns could be integrated to the toolbar menu to allow the easy recreation of patterns.

To reduce the necessary elements on the canvas a differentiation of cases should be made possible. In particular, the searching for constants can allow the generation of a set of graphs which provide relevant perspectives on Π. Such a proceeding can be compared with WHYLINE [49] for the programming language ALICE[1] where "Why did?" or "Why didn't" questions can be asked. For VIDEAS this could be reinterpreted by asking questions like "Why was a literal [not] part of this/any answer set?" or "Why was an answer set with the literals $L_1, \ldots, L_N$ [not] returned?" (or formally denoted by "Why did $\Pi \not\models^{as}$ {$[Lit_1](\ldots)$, ..., $[Lit_N](\ldots)$}?"). This could provide a backwards debugging mechanism, which builds an explanatory visualization of related elements of Π from targeted error origins.

Finally, the sketched usability enhancements require further conceptual deepening assisted by empirical user studies, and an initial practical realization. Such enhancements are crucial for reducing the visualization complexity and are therefore essential for allowing a fast recognition of (erroneous) program behaviors.

[1] Information and download: `http://www.alice.org/`, last accessed: February 24, 2011

Bibliography

[1] M. Abengozar-Carneros, P. Arenas-Sanchez, R. Caballero-Roldan, A. Gil-Luezas, J.C. Gonzalez-Moreno, J. Leach-Albert, F.J. Lopez-Fraguas, M. Rodriguez-Artalejo, J.J. Ruz-Ortiz, and J. Sanchez-Hernandez. TOY: A Multiparadigm Declarative Language. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 329–334, 2004.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.

[3] Christoph Beierle, Oliver Dusso, and Gabriele Kern-Isberner. Using Answer Set Programming for a Decision Support System. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 374–378, 2005.

[4] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme*, volume 3.0. vieweg, 2006.

[5] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. *DLV - User Manual*[2].

[6] Geraldine Brady and Todd H. Timble. A String Diagram Calculus for Predicate Logic and C.S. Peirce's System Beta. Preprint, 1998.

[7] Saartje Brockmans, Peter Haase, Pascal Hitzler, and RUdi Studer. A Metamodel and UML Profile for Rule-Extended OWL DL Ontologies. In *Proceedings of the 3rd European Semantic Web Conference*, pages 303–316. Springer, 2006.

[8] Sara Brockmans, Raphael Volz, Andreas Eberhart, and Peter Löffler. Visual Modeling of OWL DL Ontologies Using UML. In *Proceedings of the 3rd International Semantic Web Conference*, pages 198–213. Springer, 2004.

[9] Rafael Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, pages 329–334. ACM New York, 2005.

[2] `http://www.dbai.tuwien.ac.at/proj/dlv/man/`, last accessed: February 24, 2011

[10] Peter Pin-Shan Chen. The Entity-Relationship Model-Towards Unified View of Data. *Journal of ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[11] Roberto Confalonieri, Juan Carlos Nieves, and Javier Vázquez-Salceda. A Preference Meta-Model for Logic Programs with Possibilistic Ordered Disjunction. In *Proceedings of the 2nd International Workshop on Software Engineering for Answer Set Programming*, pages 19–33, 2009.

[12] Juan de Lara and Esther Guerra. Formal Support for QVT-Relations with Coloured Petri Nets. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009.

[13] G Di Battista and M Lenzerini. Deductive Entity Relationship Modeling. *Journal of IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.

[14] Thomas Eiter and Kewen WANG. Semantic Forgetting in Answer Set Programming. *Artificial Intelligence*, 172:1644–1672, 2008.

[15] H. Gallaire, J. Minker, and J.-M. Nicolas. *An Overview and Introduction to Logic and Data Bases*. Plenum Press, 1978.

[16] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A Meta-Programming Technique for Debugging Answer-Set Programs. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 1, pages 448–453. AAAI Press, 2008.

[17] Michael Gelfond. Answer Sets. In *Handbook of Knowledge Representation*. Elsevier North-Holland, Inc, 2007.

[18] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of 5h International Conference on Logic Programming*. MIT Press, 1988.

[19] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. In *New Generation Computing*, 1991.

[20] Pau Giner and Vicente Pelechano. Test-Driven Development of Model Transformations. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 748–752. Springer, 2009.

[21] Martin Gogolla. *An Extended Entity-Relationship Model: Fundamentals and Pragmatics*, volume 767 of *Lecture Notes in Computer Science*. Springer, 1994.

[22] John Grant, Tok Wang Ling, and Mong Lee. ERL: Logic for Entity-Relationship Databases. *Journal of Intelligent Information Systems*, 2(2):115–147, 1993.

[23] International Organization for Standarization. *Prolog—ISO/IEC 13211-1:1995*, 1995.

[24] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Journal of Software Tools for Technology Transfer*, 9(3):213–254, 2007.

[25] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195. ACM, 2006.

[26] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, pages 85–103. New York: Plenum, 1972.

[27] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[28] Kathrin Konczak, Torsten Schaub, and Thomas Linke. Graphs and Colorings for Answer Set Programming with Preferences. In *Fundamenta Informaticae*, volume 57. IOS Press, 2003.

[29] Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 329–334. IEEE Computer Society, 2009.

[30] Yngve Lamo. Model Driven Engineering (MDE) and Diagrammatic Predicate Logic. Technical report, Bergen University College, 2008.

[31] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. In *ACM Transactions on Computational Logic*, volume 7. ACM New York, 2006.

[32] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2004.

[33] Vladimir Lifschitz. What Is Answer Set Programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 3, pages 1594–1597, 2008.

[34] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 112–117, 2002.

[35] John McCarthy. Programs with Common Sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91. Her Majesty's Stationary Office, 1958.

[36] Ulrich Neumerkel, Christoph Rettig, and Christian Schallhart. Visualizing Solutions with Viewers. In *Proceedings of the Workshop on Logic Programming Environments*, 1996.

[37] Ilkka Niemelä and Patrik Simons. *Logic-Based Artificial Intelligence*, chapter Extending the Smodels System with Cardinality and Weight Constraints, pages 491–521. The Kluwer International Series In Engineering And Computer Science archive. Kluwer Academic Publishers, 2001.

[38] Object Management Group. *Human-Usable Textual Notation (HUTN) Specification*[3], 1.0 edition, 8 2004.

[39] Object Management Group. *Meta Object Facility (MOF) Core Specification*[4], 2006.

[40] Object Management Group. *OMG Unified Modeling Language (OMG UML)*[5], 2.2 edition, 2009.

[41] Enrico Pontelli. Answer Set Programming in 2010: A Personal Perspective. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages*, pages 1–3. Springer, 2010.

[42] Jörg Pührer. On Debugging of Propositional Answer-Set Programs. Master's thesis, 2007.

[43] Thomas Reiter. *T.R.O.P.I.C.: Transformations On Petri Nets In Color*. PhD thesis, Johannes Kepler Universität, Institute of Bioinformatics, 2008.

[44] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1-3. World Scientific Publishing, 1997.

[45] Douglas C. Schmidt. Model Driven Engineering. *Journal of IEEE Computer*, 39(2), 2006.

[46] Roy W. Schulte and Yefim V. Natis. 'Service Oriented' Architectures, Part 1. Technical report, Gartner Group, 1996.

[47] Leon Sterling. *The Art of Prolog: Advanced Programming Techniques*, volume 2 of *Series in Logic Programming*. MIT Press, 1994.

[48] 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*, volume 281 of *CEUR Workshop Proceedings*, 2007.

[49] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging*. Morgan Kaufmann and dpunkt.verlag, 2006.

[3] `http://www.omg.org/cgi-bin/doc?formal/2004-08-01`, last accessed: February 24, 2011

[4] `http://www.omg.org/spec/MOF/2.0/`, last accessed: February 24, 2011

[5] `http://www.omg.org/spec/UML/2.2/`, last accessed: February 24, 2011