FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein System für graphische Argumentationsformalismen

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Georg Heißenberger, BSc
Matrikelnummer 1026479

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Martin Diller, MSc

Wien, 16. Februar 2016

Georg Heißenberger   Stefan Woltran

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A System For Advanced Graphical Argumentation Formalisms

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Intelligence

by

Georg Heißenberger, BSc
Registration Number 1026479

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Martin Diller, MSc

Vienna, 16th February, 2016

Georg Heißenberger | Stefan Woltran

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Georg Heißenberger, BSc
Ostmarkgasse 33/2/10
1210 Wien

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

Wien, 16. Februar 2016

Georg Heißenberger

Acknowledgements

First I want to thank my advisor Stefan Woltran for his great support during the project and for his efforts to introduce me to the exciting world of GRAPPA. His tireless, patient and uncomplicated guidance through this work has been a great help to me. I also want to thank my co-advisor Martin Diller for his hands-on assistance which was an invaluable help for me.

I am also very grateful to Mario Alviano. With his deep insights into ASP and his efforts to open my eyes for the “mysteries” of saturation it became possible to formulate the static encoding. I want also to express my gratitude to Michal Morak and Manuel Bichler for their support on their Decomposition-Tool to optimize the dynamic encodings.

Finally, I want to thank Dino Rosseger. Without him I would not have attended the course which laid the foundation for this work. Moreover, he built the base for “parsing” the input of the static encoding together with his advisor Johannes Wallner.

Kurzfassung

"Argumentationstheorie" hat in den letzten Dekaden in den Computerwissenschaften immer mehr an Bedeutung gewonnen. Speziell im Bereich der "Künstlichen Intelligenz" ist es ein Gebiet, an dem rege geforscht wird. In diesem Zusammenhang hat P. M. Dung mit seiner wegweisenden Arbeit, und den darin vorgestellten "Argumentation Frameworks", den Grundstein für "Abstrakte Argumentation" gelegt. Dieses Konzept wurde vielfach aufgegriffen und für die unterschiedlichsten Anforderungen weiterentwickelt. Unter anderem ist auch GRAPPA aus dieser Entwicklung hervorgegangen und ist dabei eines der neuesten und mächtigsten Werkzeuge in diesem Bereich. GRAPPA selbst basiert auf einem Graphen dessen Kanten beschriftet sind und einer sehr ausdrucksstarken Syntax um Bedingungen für einzelne Argumente zu definieren. Diese Syntax enthält Aggregatfunktionen um die Beschriftung der eingehenden Kanten auszuwerten.

Bisher gab es für GRAPPA-Instanzen keine Werkzeuge um diese auszuwerten zu können. Darum ist es die Hauptaufgabe dieser Arbeit ein System zu entwickeln mit dem es selbst Laien leicht fällt GRAPPA-Instanzen zu erzeugen und diese auch auszuwerten. Zu diesem Zweck werden in dieser Arbeit Answer Set Programming (ASP)-Encodings zur Auswertung von GRAPPA-Instanzen vorgestellt. Außerdem wird ein Tool namens GrappaVis präsentiert. GrappaVis ist ein Programm mit dem es möglich ist intuitiv GRAPPA-Instanzen zu erstellen, diese – mit Hilfe der ASP-Programme – auszuwerten und die Ergebnisse direkt in der GRAPPA-Instanz darzustellen.

Zur Auswertung der GRAPPA-Instanzen werden zwei unterschiedliche ASP-Programme vorgestellt die auf unterschiedlichen Herangehensweisen basieren. Einerseits dem *statischen* Ansatz: Dieser basiert auf Überlegungen die auch für "Datenkomplexität" gelten, d. h. das ASP-Programm ist für alle GRAPPA-Instanzen gleich. Andererseits dem *dynamischen* Ansatz: Dieser basiert auf Überlegungen, die auf "kombinierte Komplexität" zurückgehen, d. h. das ASP-Programm zur Auswertung ist nicht mehr fix, sondern kann ebenfalls variieren. Wie sich in der Arbeit herausstellen wird, hat dieser Ansatz den Vorteil, dass man damit auch Auswertungsmethoden für GRAPPA implementieren kann, welche mit dem *statischen* Ansatz nicht realisierbar sind.

Letztendlich werden einige Resultate der ASP-Encodings bezüglich ihrer Performance präsentiert.

Abstract

Argumentation became an important research topic for computer science and in particular in Artificial Intelligence. In this field Dung introduced the landmark work of “abstract argumentation frameworks” which inspired many generalisations of this concept. Among them, GRAPPA is a very recent development and is one of the most general frameworks. The key-features of GRAPPA are, that the argumentation is based on a labeled graph and a very powerful pattern language, which includes aggregate-functions, to specify acceptance conditions for statements based on the labels of the incoming edges.

So far, there are no tools to evaluate GRAPPA-instances available. Therefore, the main goal of this work is to provide an evaluation system for GRAPPA. This system consists of two parts, namely Answer Set Programming (ASP)-encodings to evaluate GRAPPA-instances and *GrappaVis*. GrappaVis is a graphical tool to specify GRAPPA-instances, to evaluate them (making use of the ASP-encodings) and visualize the results of the evaluation.

Regarding the ASP-encodings, two different approaches are presented, namely the *static* and *dynamic* encoding to evaluate GRAPPA-instances. The *static* approach is the “classical” one, which corresponds to considerations based on *data complexity*, where one (static) program handles all GRAPPA-instances. On the other hand, the *dynamic* approach, which corresponds to considerations based on *combined complexity*, i. e. the encoding is not fixed, but dynamically generated for each GRAPPA-instance. As shown in this work this approach allows to express evaluation methods which are not expressible in the *static* approach.

Finally, performance results for the different ASP-encodings are presented.

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 Logic . . . 5
2.2 Argumentation frameworks . . . 8
2.3 Abstract dialectical frameworks . . . 9
2.4 GRAPPA . . . 11
2.5 Answer Set Programming . . . 14
2.6 Complexity . . . 29

3 Static encodings **33**
3.1 Description of the input . . . 34
3.2 States of nodes . . . 40
3.3 Basic definitions . . . 41
3.4 Model semantics . . . 51
3.5 Admissible semantics . . . 51
3.6 Complete semantics . . . 54

4 Dynamic encodings **55**
4.1 Admissible semantics . . . 55
4.2 Complete semantics . . . 60
4.3 Preferred semantics . . . 62

5 Graphical user interface **65**
5.1 Parts of the user interface . . . 66
5.2 Drawing a graph . . . 68
5.3 Handling acceptance patterns . . . 70
5.4 Evaluation of GRAPPA-instances . . . 71
5.5 Syntax of the GRAPPA pattern language for GrappaVis . . . 75
5.6 Using GrappaVis for ADFs . . . 79
5.7 Configuration of evaluation methods . . . 80
5.8 Design decisions . . . 82

6 Experimental evaluation **83**

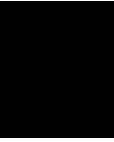
6.1 Conversion from GRAPPA to ADF 83
6.2 Conversion from ADF to GRAPPA 86
6.3 GRAPPA-instance generation 86
6.4 Performance 87

7 Conclusion and future work 91

Index 93

Bibliography 101

CHAPTER 1



Introduction

Argumentation is a natural way for humans to reach a reasonable degree of agreement on matters which are contested or not self-evident. Nevertheless, it is often quite hard to work through complex "common knowledge". Therefore, from early times on, there has been an interest in attempting to develop more structured or even formal accounts of argumentation. In fact, it can be argued that, starting with the work of the ancient Greek philosophers, this has also been one of the central concerns behind the developments of the discipline of logic.

Given the central role of argumentation in human reasoning as well as the close connection that exists between logic and computer science, it is only natural that argumentation also becomes an important research topic for computer scientists and in particular in Artificial Intelligence (AI). Today argumentation is an important sub-field of AI [BD07] not only because of the connections of argumentation with non-monotonic reasoning [SA15] but, according to [BLS14], also because of its various applications in legal reasoning [AP09], intelligent web search [CMS04, CMS07], recommender systems [CMS04, BBD$^+$12], autonomous agents and multi-agent systems [PHR$^+$11, vdWDM$^+$11] and many others [BH08, RS09, Sim11].

A significant landmark in the consolidation of the field of argumentation in AI is the work by Dung on abstract argumentation [Dun95]. Here Dung introduced what can be considered to be the first abstract argumentation formalism, now often referred as Dung's "(abstract) argumentation frameworks (AFs)". AFs provide formalisms to model concrete arguments in an abstract manner, i. e. argumentation scenarios are modeled as directed graphs where nodes represent arguments and links stand for attacks between the arguments. Dung also introduced several methods, so called "semantics", to determine the "acceptance status" of arguments based solely on the relations between arguments captured via the AFs.

To date several generalizations of Dung's AFs have been proposed to deal with more complex relations between arguments than those envisioned originally by Dung.[1]. One of the most general ones is GRAPPA, as proposed in [BW14], a formalism which is in turn based on abstract dialectical frameworks (ADFs) presented first in [BW10]. GRAPPA frameworks are labeled graphs where nodes represent statements used in arguments and "acceptance conditions", based on the labels of the incoming edges, are used to spell out the relations between the acceptance status of the statements in the framework. The outstanding feature of GRAPPA is the very powerful pattern language to define the acceptance conditions which also provides aggregates for e. g. counting labels or calculating the sum of edgelabels.

Of course there have been efforts made to develop reasoning methods to draw practical use from these frameworks but reasoning in very general frameworks suffers from high complexity. In case of ADFs it was shown in [SW14, SW15] that the reasoning tasks, depending on the semantics, are complete up to the third level of the polynomial hierarchy (Σ_3^{P}) and these results carry over to GRAPPA as shown in [BW14].

Considering these facts, it comes with no surprise that the implementation of reasoning systems for ADFs are trying to exploit powerful general purpose solver. For example DIAMOND, a reasoning tool for ADFs, is based on Answer Set Programming (ASP)-encodings [ES13]. Another approach for ADFs was proposed in [DWW14] where encodings in terms of *quantified boolean formulas (QBFs)* were developed. In both cases there are powerful solvers available.

Approaches like DIAMOND are using *static* encodings, i. e. the encoding does not change for different framework instances. Therefore, the *data complexity* of ASP has to be considered and – as shown in [EG95, EGM97] – ASP is complete in Σ_2^{P} resp. Π_2^{P}. But some evaluation methods for GRAPPA, like the preferred semantics, are complete in Σ_3^{P} and therefore it is unlikely that a static encoding for these evaluation methods can be found. However, there are ways to calculate these complex semantics, but they have limitations. For example DIAMOND evaluates an ADF-instance over the preferred semantics by splitting the problem into two consecutive sub-problems which are solved by two ASP-encodings. But the first ASP-encoding yields – in the worst case – exponential many results w. r. t. the instance size. These results are then the input for the second encoding, which thus can lead to a rather bad runtime.

GRAPPA is a very recent development, hence, no reasoning methods have been implemented yet. In this work ASP-encodings are presented to handle GRAPPA-instances. A big challenge to develop the encoding was to handle the pattern language of GRAPPA. To evaluate an instance a methodology called *saturation* is necessary for most of the semantics. But *saturation* often causes unwanted side effects which are difficult to track, especially when aggregates are used. But an essential part of the pattern language of GRAPPA are aggregates which are evaluated by ASP-aggregates in the corresponding ASP-encoding. Especially in the static encoding the application of *saturation* together

[1] See [BPW14] for a survey.

with aggregates caused much investigational efforts to circumvent the side effects and to establish a valid encoding.

But as already mentioned this approach has limitations regarding more complex semantics. Therefore, a new approach is presented in this work, namely a clever preprocessing of an instance to evaluate, which allows to write an ASP-encoding for the given instance, which *combined complexity* is one step higher in the polynomial hierarchy than in the *static* case. This provides more possibilities for the ASP-encoding. For example in case of the preferred semantics, encodings can be generated which *combined complexity* is Σ_3^P [EFFW07]. This allows the evaluation of the instance in a single solver call. This approach is called *dynamic*, because the encodings are generated for every instance individually.

Besides the encodings – static and dynamic – *GrappaVis* is presented as well, which is a tool to improve the handling of GRAPPA and ADF-instances and make the evaluation of such instances also available for ASP-laymen. *GrappaVis* is a graphical system to specify GRAPPA-instances and includes a toolbox to evaluate and compare different semantics on a generated GRAPPA-instance. With this toolbox the results of an evaluation can be easily shown directly in the graphical representation of the instance.

Summarized the main contributions of this work are:

- static ASP-encodings for GRAPPA for model, admissible and complete semantics
- dynamic ASP-encodings for GRAPPA for admissible, complete and preferred semantics
- *GrappaVis*, a graphical tool to specify and evaluate GRAPPA-instances

GrappaVis and the static encodings are available on

`http://dbai.tuwien.ac.at/proj/adf/grappavis/`

for download.

Regarding the structure of the work, in Chapter 2 the most important topics are described which are necessary to understand the main contributions of this work. This includes a formal introduction of the already mentioned frameworks, as well as the basics of ASP and the used methods – as e. g. saturation – to develop the encodings. The static encodings of GRAPPA-instances are discussed in Chapter 3, followed by the dynamic encodings in Chapter 4. In Chapter 5 the graphical user interface *GrappaVis* is presented. It also includes a tutorial on how to specify a GRAPPA-instance and how to evaluate it. A preliminary experimental evaluation and comparison of the different developed encodings can be found in Chapter 6. Finally, in Chapter 7 related and future work is discussed.

CHAPTER 2

Preliminaries

In this chapter the basic notions, systems and techniques are introduced which are necessary to understand the development of the encodings. Namely the argumentation frameworks and semantics on them are defined. To be able to compute semantics, a short introduction into classical logic and Kleene's strong three-valued logic is given first. Moreover, Answer Set Programming (ASP) is presented together with techniques how to use it.

To avoid confusion here is a short list of words which are used within this document denoting the same things:

- edge, link; in context of an argumentation framework also: dependency
- vertex, node; in context of an argumentation framework also: statement, argument

2.1 Logic

2.1.1 Two-valued logic

Two-valued logic is the classical logic.

Definition 2.1. The *syntax* is built over the alphabet given by the following items:

`Constants:` $\top, \bot$

`Connectives:` $\vee, \wedge, \rightarrow, \otimes, \neg$

`Variables:` e. g.: $p, q, r, \ldots$ ($\mathcal{V}$ denotes the set off all variables)

`Parenthesis:` (,)

Definition 2.2. A *(boolean) formula* is built inductively by the following rules:

- any constant is a formula
- any variable $v \in \mathcal{V}$ is a formula
- if A is a formula then $(\neg A)$ is a formula
- if A, B are formulas then $(A \circ B)$, $\circ \in \{\vee, \wedge, \rightarrow, \otimes\}$ is a formula

$\mathcal{L}$ denotes the set of all possible formulas.

Definition 2.3. An *(two-valued) interpretation* v^2 (v if clear from the context) is a function $v^2 : \mathcal{V} \rightarrow \{\mathbf{t}, \mathbf{f}\}$, i. e. v^2 maps to each variable $p \in \mathcal{V}$ a truth value true ($\mathbf{t}$) resp. false ($\mathbf{f}$).

A two-valued interpretation can conveniently be represented as a set of variables, i. e. $v \subseteq \mathcal{V}$. For example let $\mathcal{V} = \{p, q, r, s\}$ and $v = \{q, r\}$. This denotes the fact that $v(p) = \mathbf{f}$, $v(q) = \mathbf{t}$, $v(r) = \mathbf{t}$ and $v(s) = \mathbf{f}$.

To evaluate formulas over an interpretation, v^2 is extended to formulas:

Definition 2.4. *Extension* of v^2 to formulas: Let $a, b \in \mathcal{L}$:

$$v^2(\top) = \mathbf{t} \qquad v^2(\bot) = \mathbf{f}$$

$$v^2(\neg a) = \begin{cases} \mathbf{t} & \text{iff } v^2(a) = \mathbf{f} \\ \mathbf{f} & \text{iff } v^2(a) = \mathbf{t} \end{cases} \qquad v^2(a \wedge b) = \begin{cases} \mathbf{t} & \text{iff } v^2(a) = \mathbf{t} \text{ and } v^2(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^2(a) = \mathbf{f} \text{ or } v^2(b) = \mathbf{f} \end{cases}$$

$$v^2(a \vee b) = \begin{cases} \mathbf{t} & \text{iff } v^2(a) = \mathbf{t} \text{ or } v^2(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^2(a) = \mathbf{f} \text{ and } v^2(b) = \mathbf{f} \end{cases}$$

$$v^2(a \rightarrow b) = \begin{cases} \mathbf{t} & \text{iff } v^2(a) = \mathbf{f} \text{ or } v^2(b) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

$$v^2(a \otimes b) = \begin{cases} \mathbf{t} & \text{iff } v^2(a) = \mathbf{t} \text{ and } v^2(b) = \mathbf{f} \\ \mathbf{t} & \text{iff } v^2(a) = \mathbf{f} \text{ and } v^2(b) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

For convenience the parenthesis are omitted if the order of the operations is clear. Of course there are more operators which could be defined for this logic, but in this work it suffices to work with the negation ($\neg$), and ($\wedge$), or ($\vee$), implication ($\rightarrow$) and xor ($\otimes$). For more information about two-valued logic [KL94] can be consulted.

2.1.2 Three-valued logic

Three-valued logic is a generalization of the two-valued logic, see e. g. [Kle09].

The syntax is exactly the same as in the two-valued case given in Definition 2.1, but the semantic differs, because the interpretation is a mapping not only to true ($\mathbf{t}$) and false ($\mathbf{f}$), but also to a third value “undefined” ($\mathbf{u}$).

Definition 2.5. An *(three-valued) interpretation* v^3 (v if clear from the context) is a function $v^3 : \mathcal{V} \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

Three-valued interpretations can conveniently be represented as set of literals. A literal is a variable with or without negation. For example, consider the set of propositional variables $\{a, b, c, d, e\}$ and the three-valued interpretation $v = \{a, \neg b, d\}$. Then v represents the fact that $v(a) = \mathbf{t}$, $v(b) = \mathbf{f}$, $v(c) = \mathbf{u}$, $v(d) = \mathbf{t}$ and $v(e) = \mathbf{u}$.

Note that every two-valued interpretation is also a valid three-valued interpretation (but not vice-versa).

To evaluate formulas over an interpretation, v^3 is extended to formulas:

Definition 2.6. *Extension* of v^3 to formulas:

$$v^3(\top) = \mathbf{t} \qquad\qquad v^3(\bot) = \mathbf{f}$$

$$v^3(\neg a) = \begin{cases} \mathbf{t} & \text{iff } v^3(a) = \mathbf{f} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{t} \\ \mathbf{u} & \text{otherwise} \end{cases} \qquad v^3(a \wedge b) = \begin{cases} \mathbf{t} & \text{iff } v^3(a) = \mathbf{t} \text{ and } v^3(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{f} \text{ or } v^3(b) = \mathbf{f} \\ \mathbf{u} & \text{otherwise} \end{cases}$$

$$v^3(a \vee b) = \begin{cases} \mathbf{t} & \text{iff } v^3(a) = \mathbf{t} \text{ or } v^3(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{f} \text{ and } v^3(b) = \mathbf{f} \\ \mathbf{u} & \text{otherwise} \end{cases}$$

$$v^3(a \to b) = \begin{cases} \mathbf{t} & \text{iff } v^3(a) = \mathbf{f} \text{ or } v^3(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{t} \text{ and } v^3(b) = \mathbf{f} \\ \mathbf{u} & \text{otherwise} \end{cases}$$

$$v^3(a \otimes b) = \begin{cases} \mathbf{t} & \text{iff } v^3(a) = \mathbf{t} \text{ or } v^3(b) = \mathbf{f} \\ \mathbf{t} & \text{iff } v^3(a) = \mathbf{f} \text{ or } v^3(b) = \mathbf{t} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{f} \text{ and } v^3(b) = \mathbf{f} \\ \mathbf{f} & \text{iff } v^3(a) = \mathbf{t} \text{ and } v^3(b) = \mathbf{t} \\ \mathbf{u} & \text{otherwise} \end{cases}$$

The truth values $\mathbf{t}, \mathbf{f}, \mathbf{u}$ are ordered according to their information content by "$<_i$".

Definition 2.7. The *information ordering* $\leq_i$ over the set $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ is defined as:

$$\begin{aligned} \mathbf{u} &\leq_i \mathbf{u} & & \\ \mathbf{u} &\leq_i \mathbf{t} & \mathbf{u} &\leq_i \mathbf{f} \\ \mathbf{t} &\leq_i \mathbf{t} & \mathbf{f} &\leq_i \mathbf{f} \end{aligned}$$

It is easy to verify that $\leq_i$ induces a partial order on $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Moreover the pair $(\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}, \leq_i)$ forms a complete meet-semilattice[1] with the meet operation $\sqcap$. This meet can be read as consensus and assigns $\mathbf{t} \sqcap \mathbf{t} = \mathbf{t}$, $\mathbf{f} \sqcap \mathbf{f} = \mathbf{f}$, and returns $\mathbf{u}$ otherwise.

The information ordering can be easily extended for interpretations.

[1] A complete meet-semilattice is such that every non-empty finite subset has a greatest lower bound, the meet; and every nonempty directed subset has a least upper bound. A subset is directed if any two of its elements have an upper bound in the set.

Definition 2.8. Let $\mathcal{V}$ be a set of variables and $v_1^3, v_2^3 \subseteq \mathcal{V}$ two interpretations.

$$v_1^3 \leq_i v_2^3 \iff a \leq_i b \quad \forall a \in v_1^3, b \in v_2^3$$

Definition 2.9. A two-valued interpretation v^2 is a *completion* of a three-valued interpretation v^3 iff $v^3 \leq_i v^2$ holds.

Informally Definition 2.9 states that v^2 is basically v^3 but every $\mathbf{u}$ in v^3 is replaced by either $\mathbf{t}$ or $\mathbf{f}$.

The set of all possible completions of a three-valued interpretation v is denoted as $[v]_c$.

2.2 Argumentation frameworks

Dung's argumentation framework (AF) is not used in this work directly, but nonetheless many other frameworks are based on AFs as GRAPPA and ADFs, which are discussed in the next sections. Many notions introduced in these sections are based on definitions for AFs.

Definition 2.10. An AF is a pair $F = (AR, R)$ where A is a set of arguments, and $R \subseteq AR \times AR$ is a relation representing the conflicts ("attacks").

Definition 2.11. Let $F = (AR, R)$ be an AF, $A, B \in AR$. We say that

- A *attacks* B iff $(A, B) \in R$.
- A set S *attacks* B iff there exists an argument $C \in AR$ s. t. $(C, B) \in R$.

Definition 2.12. Given an AF $F = (AR, R)$.

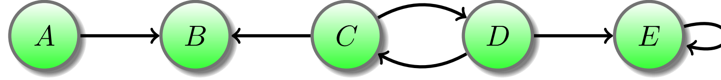
1. A set $S \subseteq AR$ is *conflict-free* in F, if, for each $a, b \in S, (a, b) \notin R$.
2. An argument $A \in AR$ is *acceptable* with respect to a set S of arguments iff for each argument $B \in AR$: if B attacks A then B is attacked by S.
3. A conflict-free set of arguments S is *admissible* iff each argument in S is acceptable with respect to S.
4. A *preferred extension* of an AF G is a maximal (w. r. t. set inclusion) admissible set of G.
5. An admissible set S of arguments is called a *complete extension* iff each argument, which is acceptable with respect to S, belongs to S.

Example 2.1. (Adapted from [Ell12]) Consider the AF $F = (AR, R)$ with

$$AR = \{A, B, C, D, E\}$$

$$R = \{(A, B), (C, B), (C, D), (D, C), (D, E), (E, E)\}$$

The following figure depicts the graphical representation of F.



For this example the following extensions exist:

- conflict free sets: $\emptyset$, $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{A, C\}$, $\{A, D\}$, $\{B, D\}$
- admissible sets: $\emptyset$, $\{A\}$, $\{C\}$, $\{D\}$, $\{A, C\}$, $\{A, D\}$
- preferred sets: $\{A, C\}$, $\{A, D\}$
- complete sets: $\{A\}$, $\{A, C\}$, $\{A, D\}$

$\triangle$

2.3 Abstract dialectical frameworks

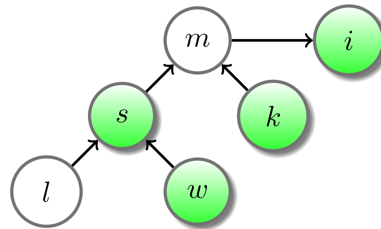
Abstract dialectical frameworks (ADFs) are a generalization of Dung-style AFs which have been first proposed in [BW10] and revisited in [BSE$^+$13].

Definition 2.13. An *abstract dialectical framework* is a tuple $D = (S, L, C)$ where

- S is a set of statements,
- $L \subseteq S \times S$ is a set of edges,
- $C = \{\phi_s\}_{s \in S}$ is a set of propositional functions over $par(s)$, one for each statement s. $par(s)$ denotes the set of parents of s, i. e. $par(s) = \{r \mid (r, s) \in L\}$. ϕ_s is called acceptance condition of s.

Example 2.2. This example is taken from [BW10] and is a variant of an example in [GPW07].

A person is innocent, unless she is a murderer. A killer is a murderer, unless she acted in self-defence. There must be evidence for self-defence, for instance a witness who is not known to be a liar. The dependency structure of the example can be represented as shown in the figure on the right side.



Assume w and k are known and l is not known, that is $\phi_w = \phi_k = \top$, $\phi_l = \bot$. The acceptance conditions for the remaining nodes are: $\phi_s = w \wedge \neg l$, $\phi_m = k \wedge \neg s$, $\phi_i = \neg m$. The shaded nodes represent the nodes which evaluate to **t** when values are propagated according to the chosen acceptance conditions. △

Definition 2.14. Let $(S, L, \{\phi_s\}_{s \in S})$ be an ADF.

A (three-valued) interpretation v is a *(three-valued) model* if for each defined[2] statement s in S holds that

$$v(s) = v(\phi_s)$$

Definition 2.15. Let D be an ADF and v a three-valued interpretation. The *characteristic operator* Γ_D is defined as

$$\Gamma_D(v)(s) = \bigsqcap \{w(\phi_s) \mid w \in [v]_c\} .$$

(for the definition of $\bigsqcap$ see Section 2.1.2)

Definition 2.16. A three-valued interpretation v for an ADF D is

1. *admissible* iff $v \leq_i \Gamma_D(v)$.
2. *complete* iff $v = \Gamma_D(v)$.
3. *preferred* iff it is $\leq_i$-maximal admissible.

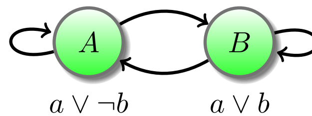
Example 2.3. Consider the ADF $D = (S, L, C)$ with

$$S = \{A, B\}, \quad C = \{\phi_A, \phi_B\}$$

$$L = \{(A, A), (B, B), (A, B), (B, A)\}$$

$$\phi_A = a \vee \neg b, \quad \phi_B = a \vee b$$

The following figure depicts the graphical representation of D.



For this example the following extensions exist:

- admissible interpretations: $\emptyset$, $\{A\}$, $\{B\}$, $\{A, B\}$, $\{\neg A, B\}$
- preferred interpretations: $\{A, B\}$, $\{\neg A, B\}$
- complete interpretations: $\emptyset$, $\{B\}$, $\{A, B\}$, $\{\neg A, B\}$

△

[2] A defined statement s is a statement where $v(s) \neq \mathbf{u}$

2.4 GRAPPA

2.4.1 Basics

GRAPPA was proposed in [BW14] as a generalization of ADF, especially by allowing more general acceptance conditions. Those acceptance conditions, informally, are defined in terms of labels which can be associated to links in GRAPPA.

Acceptance conditions of GRAPPA are defined on multisets of labels. Observe that a multiset M can be represented as a function $f_M : L \to \mathbb{N}$ where L represents the set where the elements for the multiset are taken from. For example let $L = \{+, -, *\}$ and $M = \{+, +, +, -, -\}$. The corresponding function for M is f_M which yields $f_M(+) = 3$, $f_M(-) = 2$ and $f_M(*) = 0$.

Definition 2.17. Let L be a set of labels. An acceptance function over L (L-acceptance function for short) is a function $c : (L \to \mathbb{N}) \to \{t, f\}$, that is, a function assigning a truth value to a multi-set of labels. The set of all L-acceptance functions is denoted F^L.

Definition 2.18. A labeled argument graph (LAG) is a tuple $G = (S, E, L, \lambda, \alpha)$ where

- S is a set of nodes (statements),
- E is a set of edges (dependencies),
- L is a set of labels,
- $\lambda : E \to L$ assigns labels to edges,
- $\alpha : S \to F^L$ assigns L-acceptance-functions to nodes.

2.4.2 Semantics of LAGs

Definition 2.19. Let $G = (S, E, L, \lambda, \alpha)$ be a LAG, v a three-valued interpretation of S. m_s^v, the multiset of active labels of $s \in S$ in G under v, is defined as

$$m_s^v(l) = |\{(e, s) \in E \mid e \in v, \lambda((e, s)) = l\}|$$

for each $l \in L$.

The characteristic operator Γ_G of G takes a three-valued interpretation v of S and produces a revised three-valued interpretation $\Gamma_G(v)$ of S.

Definition 2.20. Let $G = (S, E, L, \lambda, \alpha)$ be a LAG, v a three-valued interpretation of S. $\Gamma_G(v) = P_G(v) \cup N_G(v)$ with

$$P_G(v) = \{s \mid \alpha(s)(m) = t \text{ for each } m \in \{m_s^{v'} \mid v' \in [v]_c\}\}$$
$$N_G(v) = \{\neg s \mid \alpha(s)(m) = f \text{ for each } m \in \{m_s^{v'} \mid v' \in [v]_c\}\}$$

Definition 2.21. Let $G = (S, E, L, \lambda, \alpha)$ be a LAG, v a three-valued interpretation of S. Then,

- v is a model of G iff v is total and $v = \Gamma_G(v)$,
- v is grounded in G iff v is the least fixed point of Γ_G,
- v is admissible in G iff $v \subseteq \Gamma_G(v)$,
- v is preferred in G iff v is subset-maximal admissible in G,
- v is complete in G iff $v = \Gamma_G(v)$.

Example 2.4. This example is taken from [BW14]. Consider a LAG with $S = \{a, b, c, d\}$ and $L = \{+, -\}$. The graph in Fig. 2.1 shows the labels of each link.

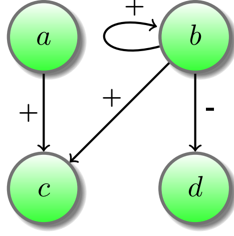


Figure 2.1: Example 2.4

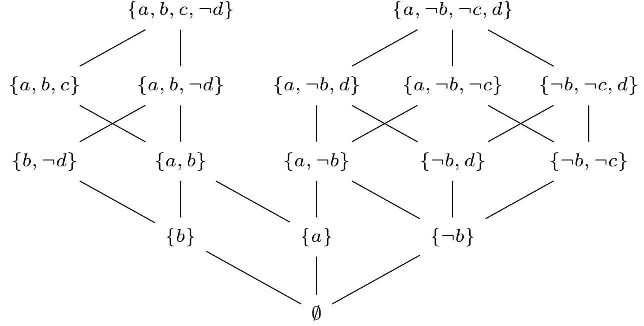


Figure 2.2: admissible interpretations of Example 2.4

For simplicity, assume all nodes have the same acceptance condition requiring that all positive links must be active (that is the respective parents must be **t**) and no negative link is active.[3] We obtain two models, namely $v_1 = \{a, b, c, \neg d\}$ and $v_2 = \{a, \neg b, \neg c, d\}$. The grounded interpretation is $v_3 = \{a\}$. 16 admissible interpretations can be obtained, shown in Fig. 2.2.

Among these admissible interpretations $\{a, b, c, \neg d\}$ and $\{a, \neg b, \neg c, d\}$ are preferred. Complete interpretations are these two and in addition $\{a\}$. △

2.4.3 GRAPPA-instances

Definition 2.22. A GRAPPA-instance is a tuple $G = (S, E, L, \lambda, \pi)$ where S, E, L and λ are as in Def. 2.19 (definition of LAGs) and

- $\pi : S \to P^L$ assigns acceptance patterns over L to nodes.

[3] In the pattern language described in Section 2.4.3 this can be expressed as

$$\#_t(+) - \#(+) = 0 \wedge \#(-) = 0$$

.

P^L here denotes the set of acceptance patterns over L defined next.

Definition 2.23. Let L be a set of labels.

- A *term* over L is of the form
  - $\#(l)$, $\#_t(l)$ for arbitrary $l \in L$,
  - $min()$, $min_t()$, $max()$, $max_t()$, $sum()$, $sum_t()$, $count()$, $count_t()$.
- A *basic acceptance pattern* (over L) is of the form

$$a_1 t_1 + \cdots + a_n t_n R\, a$$

  where the t_i are terms over L, the a_i's and a are integers and $R \in \{<, \leq, =, \neq, \geq, >\}$.
- An *acceptance pattern* (over L) is a basic acceptance pattern or a boolean combination of acceptance patterns.

2.4.4 Semantics of GRAPPA-instances

Definition 2.24. Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance. For a multiset of labels $m : L \to \mathbb{N}$ and $s \in S$ the value function val_s^m is defined as:

$$\begin{array}{ll}
val_s^m(\#l) & = m(l) \\
val_s^m(\#_t l) & = |\{(e,s) \in E \mid \lambda((e,s)) = l\}| \\
val_s^m(min) & = \mathbf{min}\, m \\
val_s^m(min_t) & = \mathbf{min}\{\lambda((e,s)) \mid (e,s) \in E\} \\
val_s^m(max) & = \mathbf{max}\, m \\
val_s^m(max_t) & = \mathbf{max}\{\lambda((e,s)) \mid (e,s) \in E\} \\
val_s^m(sum) & = \sum_{l \in L} m(l) \\
val_s^m(sum_t) & = \sum_{(e,s) \in E} \lambda((e,s)) \\
val_s^m(count) & = |\{l \mid m(l) > 0\}| \\
val_s^m(count_t) & = |\{\lambda((e,s)) \mid (e,s) \in E\}|
\end{array}$$

$min_{(t)}$, $max_{(t)}$, $sum_{(t)}$ are undefined in case of non-numerical labels. In case of an emptyset they yield the neutral element of the corresponding operation, i. e.[4]

$$\begin{aligned}
val_s^m(sum) = val_s^m(sum_t) = {} & 0 \\
val_s^m(min) = val_s^m(min_t) = {} & \infty \\
val_s^m(max) = val_s^m(max_t) = {} & -\infty
\end{aligned}$$

[4] In this point the definition differs to the original paper [BW14], where these cases are left undefined.

Definition 2.25. Let m be a multiset of labels and s a statement. The *satisfaction relation* $\models$ is defined

- for basic acceptance patterns :

$$(m, s) \models a_1 t_1 + \cdots + a_n t_n R\, a \quad \text{iff} \quad \sum_{i=1}^{n} \left(a_i\ val_s^m(t_i)\right)\ R\ a.$$

  $R \in \{<, \leq, =, \neq, \geq, >\}$

- for boolean operators as usual. For example consider the following rules: Let p_1, p_2 acceptance conditions.

$$\begin{aligned}
(m, s) \models \neg p_1 \quad & \text{iff } (m, s) \not\models p_1 \\
(m, s) \models p_1 \wedge p_2 \quad & \text{iff } (m, s) \models p_1 \text{ and } (m, s) \models p_2 \\
(m, s) \models p_1 \vee p_2 \quad & \text{iff } (m, s) \models p_1 \text{ or } (m, s) \models p_2 \\
(m, s) \models p_1 \otimes p_2 \quad & \text{iff } (m, s) \models p_1 \text{ and } (m, s) \not\models p_2 \text{ or} \\
& \quad (m, s) \not\models p_1 \text{ and } (m, s) \models p_2
\end{aligned}$$

With these definitions the connection with LAGs can be established: For each node s the function $\alpha(s)$ is linked to the corresponding pattern $\pi(s)$ as follows:

$$\alpha(s)(m) = \mathbf{t} \text{ iff } (m, s) \models \pi(s).$$

Now the characteristic operator for a GRAPPA-instance can be defined equivalently to LAGs.

Definition 2.26. Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance. The *characteristic operator* $\Gamma_G(v) = P_G(v) \cup N_G(v)$ with

$$\begin{aligned}
P_G(v) &= \left\{ s \mid (m, s) \models \pi(s) \text{ for each } m \in \left\{ m_s^{v'} \mid v' \in [v]_c \right\} \right\}, \\
N_G(v) &= \left\{ \neg s \mid (m, s) \not\models \pi(s) \text{ for each } m \in \left\{ m_s^{v'} \mid v' \in [v]_c \right\} \right\}.
\end{aligned}$$

With the operator Γ_G from Definition 2.26 the semantics of LAGs from Definition 2.21 can be carried over to GRAPPA-instances.

2.5 Answer Set Programming

Answer Set Programming (ASP) [PS01, GL02, Lif02, MT99, Nie99] is a declarative problem solving paradigm which has its roots in logic programming and non-monotonic reasoning. In contrast to the procedural approach no algorithms are defined to solve a certain problem, but a set of rules is specified and a solver calculates the solutions for the given problem. In this section a brief introduction to ASP is given and covers only topics which are necessary to understand the encodings which are presented later in this

work. The introduction is based on the features of *Potassco* – described in [GKK+11] – and therefore may not correspond exactly to the ASP-standard. A more profound introduction can be found in [EIK09] and a comprehensive treatise about ASP is given in [GKKS12].

2.5.1 Basics

In this section the syntax for ASP is introduced and analogies between ASP and classical logic are pointed out.

An ASP-program is a finite set of rules where a rule in general consists of two parts, the *head* and the *body* of the rule.

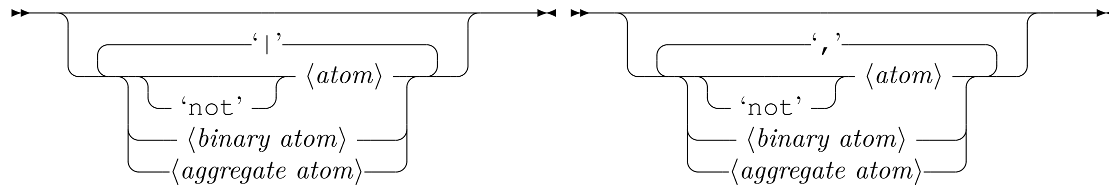
rule:

⟨*head*⟩ – ‘`:-`’ – ⟨*body*⟩ – ‘`.`’

A rule is basically an implication and “`:-`” is a ASCII-abstraction of the – mirrored – implication arrow $\leftarrow$.

The head and the body of the rule are syntactically the same and are just a list of different kinds of atoms.

head: **body:**



The major difference is the semantics of atoms occurring in the head and the body of the rule. The atoms of the body are implicitly within a conjunction and the atoms of the head are within a disjunction, i. e.

$$\texttt{a}_1\texttt{|} \ \ldots \ \texttt{|a}_\texttt{n} \texttt{ :- } \texttt{b}_1\texttt{,} \ \ldots \ \texttt{,b}_\texttt{k}\texttt{.}$$

corresponds to the boolean formula

$$a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k.$$

It is said that a rule *fires* if the body of the rule is true. If a rule fires, it *derives* its head.

Depending on the structure of the rule different types of rules are classified. A rule

$$\texttt{a}_1\texttt{|} \ \ldots \ \texttt{|a}_\texttt{n} \texttt{ :- } \texttt{b}_1\texttt{,} \ \ldots \ \texttt{,b}_\texttt{k}\texttt{, not b}_{\texttt{k+1}}\texttt{,} \ \ldots \ \texttt{,not b}_\texttt{m}\texttt{.}$$

is called

- a *disjunctive rule*,

- a *normal rule* if $n \leq 1$, i. e. there is at most one atom in the head of the rule,
- a *positive rule* if $k = m$, i. e. the keyword "`not`" does not appear in the rule,
- a *fact* if $m = 0$, i. e. the rule has an empty body. The empty conjunction is always true and therefore the head must be true to satisfy the rule. "`:-`" is usually omitted.

$$\mathtt{a_1 | \ldots | a_n.}$$

- a *constraint* if $n = 0$, i. e. the rule has an empty head. Because an empty disjunction is always false the body of the rule must also be false in order to satisfy the rule.

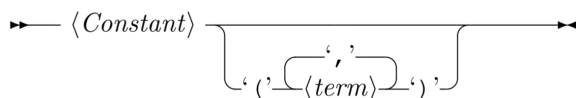
$$\mathtt{:- b_1, \ldots , b_m.}$$

What exactly *true* and *false* means in this context is described in Section 2.5.5.

This classification carries over to ASP-programs: An ASP-program is called *disjunctive* if *disjunctive* rules are included. The program is called *normal* if only *normal* rules are included.

An atom is basically representing an n-ary predicate. A predicate with no arguments is a propositional variable.

atom:

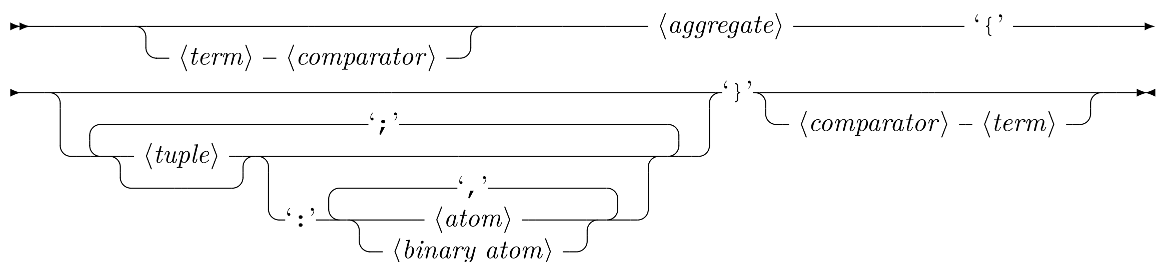


A binary atom is just the infix notation for some "built-in" binary predicate (predicates the solver provides).

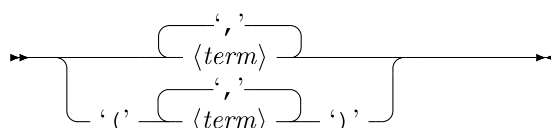
binary atom:

An *aggregate atom* is providing functionality to evaluate a set of *tuples*, i. e. to get the number of tuples in the set, to calculate the sum of all tuples or to retrieve the minimum/ maximum of the set. For calculating the sum, minimum and maximum only the first element of the tuples is considered by the solver.

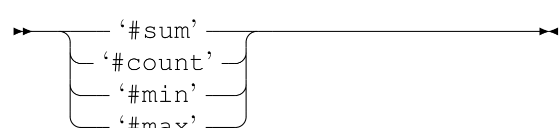
aggregate atom:



tuple:



aggregate:



It is very important to be aware that these functions work over sets. For example consider the rules in Listing 2.1.

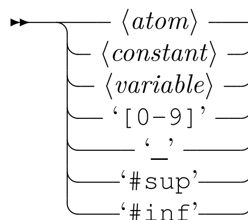
```
1 a :- 1 = #sum{1;1}.
2 b :- 2 = #sum{1,a;1,b}.
```

Listing 2.1: example for aggregates

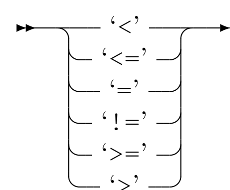
The program derives a and b. The aggregate of the first line yields 1 because in a set 1 only appears once. For the aggregate in the second line the elements of the set can be distinguished by the second element of the tuple. Therefore, the result of the second aggregate is 2.

Another point to take care of is the case when the set to evaluate is empty. In this case the aggregate yields the neutral element of the corresponding operation. For `#sum` and `#count`-aggregates this is just zero. But `#min` returns `#sup`, i. e. ∞, and `#max` returns `#inf`, i. e. $-\infty$, for an empty set.

term:



comparator:



The keywords `#sup` represents ∞ and `#inf` stands for $-\infty$.

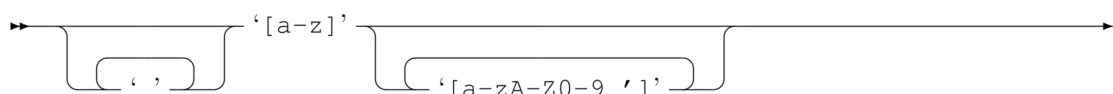
Noteworthy for *term* is that the underscore "_" is used for arguments within predicates which do not matter in a specific rule, e. g. consider Listing 2.2

```
1 child(george, 8).
2 child(john, 11).
3
4 age(X) :- child(_, X).
```

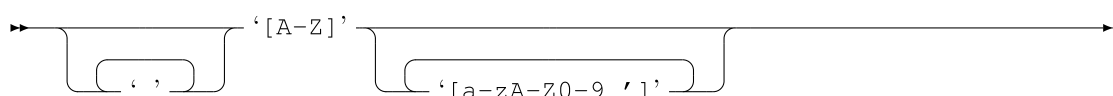
Listing 2.2: example for the use of an underscore

The difference between a *constant* and a *variable* is only the first letter. A *constant* must start with a lower-case letter whereas the first letter of a *variable* is upper-case.

constant:



variable:



2.5.2 Grounding

The process of replacing all variables from an ASP-program by adequate constants is called *grounding*. *Grounding* is usually done by a separate program, the *grounder*.

For example consider Listing 2.3. In the last line `X` is a variable which is replaced by every possible value for `X`, in this case $\{0, 1, 2\}$, and results in the program given in Listing 2.4.

```
a(0).
a(1).
a(2).

b(X) :- a(X).
```

Listing 2.3: example for grounding

```
a(0).
a(1).
a(2).

b(0) :- a(0).
b(1) :- a(1).
b(2) :- a(2).
```

Listing 2.4: example after grounding

Depending if an atom, a rule or a program includes variables they are called *ground*, or if not, *non-ground* atom, rule or program.

Definition 2.27.
A *ground atom* is an atom with no variable.
A *ground rule* is a rule with no variable.
A *ground program* is a program with no variable.

One important aspect of grounding is that the variables need to be *safe*. *Safe variables* are variables where the grounder can infer the values the variable can take. Variable `X` in the last line of Listing 2.3 is a *safe variable*. In contrary the variable `X` in the rule in Listing 2.5 is *unsafe*. The grounder can not infer which values the variable `X` can take. In the default settings the grounder stops in case of an *unsafe variable* and returns an error.

```
a(X) :- b.
```

Listing 2.5: example of an unsafe variable

In general, grounding is much more complex than it looks in these easy examples, but for the purpose of this work the given information suffices. For further reference please consult [GKKS12].

2.5.3 Intervals

Intervals are syntactic-sugar to abbreviate the notation of – for example – many facts.

Consider the case that atoms **intv(X)** need to be defined for $21 \leq X \leq 53$. One possibility would be to define every single atom, what is very tedious for obvious reasons. The much more convenient way is to use the interval notation where the start and the

end of the interval are separated by two points "`..`". This way – for the definition of the atoms **intv** – it suffices to state:

```
intv(21..53).
```

As already mentioned this notation is only an abbreviation. There is no semantic difference between the interval notation and stating every single atom. In fact, the grounder instantiates the atom for every single element anyway, so it is just less to write for the author of the encoding.

2.5.4 Default negation

The keyword "`not`" denotes the so called *default negation*[5] and can be placed in front of an atom. The *default negation* is also referred to as "negation as failure" because `not a` becomes true if `a` is not derived by any other rule of the program.

```
d :- not e.
```

Listing 2.6: example for default negation

For example consider Listing 2.6. `d` is derived, because no other rule derives `e`.

Observe that the *default negation* does behave differently to the classical negation.[6] If the rule from Listing 2.6 would be translated to

$$\neg e \rightarrow d$$

there would be three classical (two-valued) models of the rule, namely $\{d\}$, $\{e\}$ and $\{e, d\}$. But especially the model $\{e, d\}$ does not satisfy the intention of the *default negation*, because if `e` is derived the rule from Listing 2.6 may not derive `d`.

Among other things this gives rise to the *stable model semantics*.

2.5.5 Stable model semantics

The constructs of ASP are closely related to classical logic and therefore it seems obvious to define a semantics of ASP based on classical logic. But the default negation obstructs a direct approach because it behaves differently to the classical negation. In Listing 2.6 already a small example was given.

To overcome the troubles induced by the *default negation* the *stable model semantics* is used which makes use of the Gelfond-Lifschitz reduct [GL88]. To obtain this reduct, basically all *default negations* are removed from the program, which makes it possible to evaluate the rules of the program as boolean formulas.

[5] In ASP there is also a *strong negation* but it is not used in the encodings presented in this work.

[6] Refer to the surveys [Bid91, AB94] and to [EIP+06, EIKP08] for more discussion.

Definition 2.28. Let P be a ground ASP-program and $\mathcal{A}_P$ the set of all ground atoms of P. An *interpretation* I is a subset of $\mathcal{A}_P$.

Informally I includes all those ground atoms which are assumed to be true.

Definition 2.29. The Gelfond-Lifschitz reduct (or simply reduct) of a program P w. r. t. an interpretation M, denoted P^M, is a program obtained by

1. removing rules with "`not a`" in the body for each $a \in M$ and
2. removing default negated atoms "`not a`" from all other rules.

Definition 2.30. Let P be a program and M an interpretation. An interpretation N is a *model* of the reduct P^M if N is a classical (two-valued) model of all rules in P^M.

Definition 2.31. Let P be a program and M an interpretation. A model N of a reduct P^M is *minimal*, if there exists no model J of P^M s. t. $J \subset N$.

$LM(P^M)$ denotes the set of all minimal models of P^M.

Definition 2.32. An interpretation M of P is a *stable model* of P, if

$$M \in LM(P^M).$$

In this context often the term *candidate model* is used. A *candidate model* of a program P is just an interpretation M which is then used to calculate the reduct P^M.

```
a :- not b.
b :- not a.
```

Listing 2.7: example stable model

For example consider Listing 2.7: There are the four candidate models `{},{a},{b},{a,b}`. The corresponding reducts are given in Listings 2.8 to 2.11

```
a.
b.
```

Listing 2.8: reduct $P^{\{\}}$

```
a.
```

Listing 2.9: reduct $P^{\{a\}}$

```
b.
```

Listing 2.10: reduct $P^{\{b\}}$

```
```

Listing 2.11: reduct $P^{\{a,b\}}$

So according to Definition 2.32:

$$\{\} \neq \{a,b\} \in LM\left(P^{\{\}}\right) \qquad \{b\} \in LM\left(P^{\{b\}}\right)$$
$$\{a\} \in LM\left(P^{\{a\}}\right) \qquad \{a,b\} \neq \{\} \in LM\left(P^{\{a,b\}}\right)$$

Therefore, the example in Listing 2.7 has only two stable models, namely `{a}` and `{b}`.

In this context stable models of a given ASP-program are also called *answer-sets*.

2.5.6 Guess and check

Guess and check is an important methodology used in ASP. As the name suggests it is based on a guessing part, where candidate solutions are generated. Then these guesses are checked against constraints which filter out the candidate solutions which are incorrect.[7]

How this paradigm could be used to determine whether a graph can be colored using three colors is shown in Example 2.5.

Example 2.5. Let $G = (V, E)$ be an undirected graph. Check whether a color $c \in \{r, g, b\}$ can be assigned to each vertex $v \in V$ s. t. there are no two adjacent vertices with the same color, i. e. for every undirected edge $\{s, t\} \in E : c(s) \neq c(t)$.

An instance of a graph is encoded with the predicate **s(v)** for each $v \in V$ and **e(x,y)** for each $\{x, y\} \in E$ as shown in Listing 2.12.

```
1 s(a). s(b).
2 s(c). s(d).
3
4 e(a,b).
5 e(b,d).
6 e(a,c).
```

Listing 2.12: an example instance for three-colorability

```
1 r(X)|g(X)|b(X) :- s(X).
2
3 :- e(X,Y), r(X), r(Y).
4 :- e(X,Y), g(X), g(Y).
5 :- e(X,Y), b(X), b(Y).
```

Listing 2.13: a program using the guess and check methodology for three-colorability

The guess and check is performed in Listing 2.13. Line 1 expresses the guess: Every vertex can potentially be colored using any of the three colors. The check is done by the constraints in Lines 3 to 5. The rules remove every color assignment where the vertices with the same color are adjacent in the graph. The results of the program are all valid color assignments of the graph. △

2.5.7 Subprograms

Subprograms are a well-known methodology in procedural programming languages which helps to reuse existing parts of code. Although for ASP, subprograms are not available, there is a possibility to reuse existing code. For example consider the program from Listing 2.13. Assume that it is necessary to do one (or even more) new guess(es) of the three colors, and the constraints in Lines 3 to 5 are also required for the new guess(es).

The naive approach would be to just copy paste the rules and rename the predicates to meet the new requirement, but this not very convenient.

To encapsulate the existing rules into some kind of subprogram is a more elegant solution. This can be achieved by adding a new argument to all predicates which are involved

[7] A more elaborate description of this technique can be found in [EFLP00].

in the subprogram. This new argument is then used to address the “instance” of the subprogram.

The adapted program of Listing 2.13 is shown in Listing 2.14 and can be considered as a subprogram. The new atom **guess(I)** in the guessing-rule, in Line 3, is necessary to “trigger” the different “calls”. Every fact **guess(X)** corresponds to a new “call” of the subprogram, i. e. a new guess with “instance-id” `X` is done and the invalid interpretations of the guess are removed by the constraints in Lines 5 to 7. In case of the atoms given in Line 1 the subprogram is “executed” three times with the “instance-ids” $\{a, 1, 0\}$.

```
1 guess(a). guess(1). guess(0).
2
3 r(I,X)|g(I,X)|b(I,X) :- s(I,X), guess(I).
4
5 :- e(X,Y), r(I,X), r(I,Y).
6 :- e(X,Y), g(I,X), g(I,Y).
7 :- e(X,Y), b(I,X), b(I,Y).
```

Listing 2.14: guess and check of three-colorability as subprogram

2.5.8 Encode NP-complete problems into one rule-body

An idea which is used for the dynamic encodings – which are presented in Chapter 4 – is to encode an NP-complete problem into a single rule.

For example consider the three-colorability of a graph which is a well-known NP-complete problem. The “classical” encoding was already shown in Listing 2.13. To encode the three-colorability of a graph $G = (V, E)$ into one rule, six facts are necessary, because the program must know all valid color-variations of a single edge. This information is defined with the predicate **validColoring** in the Eqs. (2.1) to (2.3).

With these facts a single rule can be defined to check whether a valid coloring of the graph is possible. W. l. o. g. assume that the nodes in V are represented as integer values. Let

$$\psi := \bigwedge_{(i,j) \in E} validColoring(X_i, X_j)$$

The complete program is given in Eqs. (2.1) to (2.4)

$$validColoring(red, blue).\ validColoring(red, green). \tag{2.1}$$

$$validColoring(green, red).\ validColoring(green, blue). \tag{2.2}$$

$$validColoring(blue, red).\ validColoring(blue, green). \tag{2.3}$$

$$valid \leftarrow \psi. \tag{2.4}$$

So the ASP-solver is testing all possibilities to assign a color to the X_i variables for all $i \in V$. But the rule in Eq. (2.4) fires only if at least one valid assignment has been found. The program yields an answer-set anyway, but the answer-set includes the atom **valid** only if the given graph is three-colorable.

2.5.9 Saturation

Sometimes there are problems associated to the question if all answer-sets exhibit a certain property. At a first glance it is impossible to define an ASP-program which can give an answer to such a question, because the rules only work “within” one interpretation and can not “access” any other interpretation. But there is a trick – called *saturation* – to achieve such a behavior.

Example 2.6. As in Example 2.5, let $G = (V, E)$ be an undirected graph. Consider the problem to determine whether the graph is not three-colorable.

Of course the program from Example 2.5 can be used: If there is no answer-set the graph is not three-colorable. But what if the program should yield an answer-set in this case? A first approach could look like the program in Listing 2.15. But this program yields an answer-set for every invalid coloring.

But only one answer-set which states that the graph is not three-colorable is wanted. Consider Listing 2.16, where the saturation technique is applied. This technique is called “saturation” because if the property – in this case no valid three-coloring – holds, all color-defining atoms – **r(X)**, **g(X)**, **b(X)** – are saturated into the answer-set, which is done in the Lines 7 to 9.

```
1 r(X)|g(X)|b(X) :- s(X).
2
3 invalid :- e(X,Y), r(X), r(Y).
4 invalid :- e(X,Y), g(X), g(Y).
5 invalid :- e(X,Y), b(X), b(Y).
6
7 :- not invalid.
```

Listing 2.15: non three-colorability without saturation

```
1 r(X)|g(X)|b(X) :- s(X).
2
3 invalid :- e(X,Y), r(X), r(Y).
4 invalid :- e(X,Y), g(X), g(Y).
5 invalid :- e(X,Y), b(X), b(Y).
6
7 r(X) :- invalid, s(X).
8 g(X) :- invalid, s(X).
9 b(X) :- invalid, s(X).
```

Listing 2.16: non three-colorability with saturation

△

The saturation allows all interpretations – which exhibit the property of interest – to share the same answer-set, namely the set of all atoms affected by the saturation. So if all interpretations have the property, the program yields only this “saturated” answer-set. If there is any interpretation which does not exhibit the property of interest, the program yields only the answer-set which does not exhibit the property of interest. This answer-set is a subset of the saturated answer-set and because answer-sets are subset-minimal the saturated answer-set is no longer a valid stable model.[8] Finally, to remove all invalid answer-sets the rule

```
:- not invalid
```

can be added to the program of Listing 2.16.

[8] For a more elaborate explanation please consult [EIK09].

Dealing with aggregates

Using aggregates together with saturation is anything but easy, because the saturation can affect the set the aggregate is evaluating. Therefore, the result of the aggregate can be different before and after saturation. Moreover, the reduct of the program removes rules that do not fire.[9] This leads to unexpected results.

Example 2.7. Let S be a subset of the natural numbers, i. e. each element $n \in S : n \geq 0$. Verify that for every non-empty subset $\emptyset \neq R \subseteq S$ holds that

$$\min_{r \in R} r \leq \sum_{r \in R} r.$$

Observe that the condition is trivially true for every subset of the natural numbers. Anyway, an ASP-program should be written to confirm the statement for a given set of natural numbers.

In Listing 2.17 an ASP-encoding is given to solve the task without saturation. The program expects an input of the set of natural numbers S encoded as `s(`n`)` for every $n \in S$. The input in Line 1 of Listing 2.17 corresponds to a set $S = \{23, 30, 123\}$.

```
1 s(23).s(30).s(123).
2
3 in(X) | out(X) :- s(X).
4 cnt(X) :- X = #count{Y:in(Y)}.
5
6 min(Min) :- Min = #min{X:in(X)}.
7 sum(Sum) :- Sum = #sum{X:in(X)}.
8
9 ok :- cnt(0).
10 ok :- min(Min), sum(Sum), Min <= Sum.
```

Listing 2.17: approach for Example 2.7 without saturation

The program guesses all possible subsets of S in Line 3 and the atom **in(X)** identifies all numbers of the subset. In Lines 6 and 7 the minimum and the sum of the subset is calculated and in Line 10 the values are compared. Line 9 is necessary because in case of the empty subset the atom **ok** is simply added to the answer-set. △

The approach in Listing 2.17 has a big disadvantage, namely that it yields an answer-set for every subset. This is means that $2^{|S|}$ answer-sets have to be checked whether the atom **ok** is part of the answer-set. It would be much more convenient if there would be only one answer-set which includes **ok** only if the condition is true.

With saturation all answer-sets of the program in Listing 2.17 can be merged into one, but to apply saturation special measures have to be taken. The atom **ok** is used to "trigger"

[9] If aggregates are used in ASP-programs the stable model semantics does not suffice. Therefore, other semantics are used which remove rules with unsatisfied body. More information can be found in [FPL11].

saturation and to derive all atoms depending on the guess in Line 3, namely **in**, **out**, **cnt**, **min** and **sum**. For **in**, **out** it is easy, just add the rules as shown in Listing 2.18

```
1 in(X) :- ok, s(X).
2 out(X) :- ok, s(X).
```

Listing 2.18: saturation rules for **in** and **out**

```
1 dom(0..200).
2 cnt(X) :- dom(X), ok.
3 min(X) :- dom(X), ok.
4 sum(X) :- dom(X), ok.
```

Listing 2.19: naive saturation rules for **cnt**, **min**, **sum**

For **cnt**, **min** and **sum** also all possible atoms need to be saturated. To facilitate the task it is convenient to introduce a *domain-predicate* which defines the values which may occur. Of course the domain must be chosen according to the current instance, but as small as possible to keep the set of atoms small. So a first approach for the saturation rules could look like the rules in Listing 2.19 where **dom** is the domain-predicate. But the program – all rules from Listings 2.17 to 2.19 together – is not working. The solver claims that the program is unsatisfiable.

The reason for that behavior is, that in the reduct of the program some important rules are removed. Let the candidate model M the set of all atoms – correctly saturated – and P the program given by the Listings 2.17 to 2.19. Consider the reduct P^M shown in Listing 2.20 and especially the Lines 4, 6 and 7. It is easy to verify that the interpretation $I = \{$`s(23),s(30),s(123),out(23),out(30),out(123), dom(0..200)`$\}$ is a model of P^M and therefore is M no stable model of P.

```
1 s(23).s(30).s(123).
2
3 in(X) | out(X) :- s(X).
4 cnt(3) :- 3 = #count{Y:in(Y)}.
5
6 min(23) :- 23 = #min{X:in(X)}.
7 sum(176) :- 176 = #sum{X:in(X)}.
8
9 ok :- cnt(0).
10 ok :- min(Min), sum(Sum), Min <= Sum.
11
12 in(X) :- ok, s(X).
13 out(X) :- ok, s(X).
14
15 dom(0..200).
16 cnt(X) :- dom(X), ok.
17 min(X) :- dom(X), ok.
18 sum(X) :- dom(X), ok.
```

Listing 2.20: reduct P^M

To avoid the removal of rules in the reduct, the rules with aggregates have to be reformulated in a way that the rule itself performs the saturation as well. Consider a rule

with a #sum-aggregate as shown in Line 1 in Listing 2.21. Such a rule can be rewritten as shown in Line 3. The atom **ok** again is the trigger for saturation and **dom** is the domain-predicate defining all possible values for X. If no saturation is performed the new rule just sums up all elements of **p**. If saturation should be done, the trick is to eliminate each element $a \in \mathbf{p}$ by adding the inverse element $-a$ which causes the sum to be zero. Finally, X is added to the set such that the #sum-aggregate yields X as result.

For rules with a #count-aggregate the conversion is similar, because a #count-aggregate can easily be rewritten as a #sum-aggregate. The rules in Lines 1 and 2 in Listing 2.22 are equivalent, therefore the reformulation of sum can be applied on the rule in Line 2 and yields the rule in Line 4.

```
1 sum(X) :- X = #sum{A:p(A)}.
2
3 sum(X) :- dom(X),
4           X = #sum{ A: p(A);
5                    -A: p(A), ok;
6                     X,ok: ok}.
```

Listing 2.21: reformulation of #sum

```
1 count(X) :- X = #count{A:p(A)}.
2 count(X) :- X = #sum{1,A:p(A)}.
3
4 count(X) :- dom(X),
5             X = #sum{ 1, A: p(A);
6                      -1, A: p(A), ok;
7                       X,ok: ok}.
```

Listing 2.22: reformulation of #count

To find a reformulation for #min/#max-aggregates they need to be rewritten as a #sum-aggregate. This can be achieved by the Eqs. (2.5) and (2.6) as shown in [AFG15].

$$min[w_1 : l_1, \dots, w_n : l_n] = b \equiv sum[1 - n \cdot (b - w_i) : l_i \mid i \in [1 \dots n], w_i \leq b] > 0 \tag{2.5}$$

$$max[w_1 : l_1, \dots, w_n : l_n] = b \equiv sum[1 + n \cdot (b - w_i) : l_i \mid i \in [1 \dots n], -w_i \leq -b] > 0 \tag{2.6}$$

Therefore, a rule with a #min-aggregate – as shown in Line 1 of Listing 2.23 – can be reformulated to an equivalent rule using Eq. (2.5). But this reformulation requires the number of elements – n – of the set under evaluation. This number is retrieved by the rule in Line 3. Hence, the rule in Line 4 is equivalent to the rule in Line 1. In order to ensure that the rule also fires in case of saturation, a big value is added – in Line 14 – to achieve that the result of #sum is larger than zero in case of saturation.

Finally, the rule in Line 3 has to be considered, because the rule is not yet safe for saturation. To this end the already established reformulation of #count-aggregates has to be applied to the rule. So the reformulation of a #min-aggregate yields two rules shown in the Lines 7 and 11 of Listing 2.23.

The reformulation of a rule with a #max-aggregate – as in Line 1 of Listing 2.24 – is the same as for a #min-aggregate, just Eq. (2.6) has to be used. The reformulation again yields the two rules given in Lines 3 and 7.

```
1  min(X) :- X = #min{A:p(A)}.
2
3  cnt(X) :- #count{A:p(A)}.
4  min(X) :- dom(X), cnt(N),
5        0 < #sum{1-N*(X-A):p(A),A<=X}.
6
7  cnt(X) :- dom(X),
8            X = #sum{ 1, A: p(A);
9                     -1, A: p(A), ok;
10                     X,ok: ok}.
11 min(X) :- dom(X),cnt(N),
12       0 < #sum{
13             1-N*(X-A):p(A),A<=X;
14             100000: ok}.
```

Listing 2.23: reformulation of `#min`

```
1  max(X) :- X = #max{A:p(A)}.
2
3  cnt(X) :- dom(X),
4            X = #sum{ 1, A: p(A);
5                     -1, A: p(A), ok;
6                      X,ok: ok}.
7  max(X) :- dom(X),cnt(N),
8        0 < #sum{
9              1+N*(X-A):p(A),-A<=-X;
10             100000: ok}.
```

Listing 2.24: reformulation of `#max`

Example 2.8. (continued from Example 2.7)

With the reformulations of the aggregates an ASP-program can be defined which yields indeed only one answer-set.

```
1  s(23).s(30).s(123).
2  dom(0..200).
3
4  in(X) | out(X) :- s(X).
5
6  cnt(X) :- dom(X), X = #sum{1,Y:in(Y);
7                            -1,Y:in(Y),ok;
8                             X: ok}.
9  min(Min) :- dom(Min), cnt(N), 0 < #sum{ 1-N*(Min-X):in(X), X<=Min;
10                                        10000: ok}.
11
12 sum(Sum) :- dom(Sum), Sum = #sum{X:in(X);
13                                 -X:in(X),ok;
14                                  Sum,ok:ok}.
15 ok :- cnt(0).
16 ok :- min(Min), sum(Sum), Min <= Sum.
17
18 in(X) :- s(X), ok.
19 out(X) :- s(X), ok.
```

Listing 2.25: approach for Example 2.7 with saturation

△

Missing answer-sets

In Listing 2.26 a small program is presented, where the rule in Line 7 fires if there are strictly more **in** than **out** atoms. By setting one of the two possible statements to **in** this is trivially true. Therefore, we get two answer-sets, namely `{s(a), s(b), in(a), in(b) ok}` and `{s(a), s(b), in(a), out(b), ok}` as expected. Just for

demonstration saturation should be done if **ok** is derived to get just one answer-set instead of two. To achieve that behavior the rules – given in the Lines 11 and 12 of Listing 2.27– have to be added. These rules derive all **in**/ **out** atoms for each **s** in case of saturation.

```
1 s(a).
2 s(b).
3
4 in(a).
5 in(X) | out(X) :- s(X).
6
7 ok :- 0 < #sum{1;
8               1,X:in(X);
9              -1,X:out(X);
10             10:ok}.
```

Listing 2.26: example without saturation

```
1 s(a).
2 s(b).
3
4 in(a).
5 in(X) | out(X) :- s(X).
6
7 ok :- 0 < #sum{1;
8               1,X:in(X);
9              -1,X:out(X);
10             10:ok}.
11 in(X) :- s(X), ok.
12 out(X) :- s(X), ok.
```

Listing 2.27: example with saturation

But the solver now does not yield the expected single answer-set but claims that this program is unsatisfiable. To understand this behavior the program and its candidate models have to be analyzed. The obvious candidate model is `{s(a), s(b), in(a), in(b), out(a), out(b), ok}` but there is also a second one `{s(a), s(b), in(a), out(a), out(b)}`. The reducts for those candidates are shown in the Listings 2.28 and 2.29.

```
s(a).
s(b).

in(a).
in(X) | out(X) :- s(X).

ok :- 0 < #sum{1,X:in(X);
    ↪ -1,X:out(X); 10:ok}.

in(X) :- s(X), ok.
out(X) :- s(X), ok.
```

Listing 2.28: reduct for candidate 1

```
s(a).
s(b).

in(a).
in(X) | out(X) :- s(X).
```

Listing 2.29: reduct for candidate 2

In the first case there is `{s(a), s(b), in(a), out(a), out(b)}` and for the second case is `{s(a), s(b), in(a), out(b)}` a smaller model than the corresponding candidate. So neither one of the candidates is a stable set and therefore the program is unsatisfiable.

Informally speaking the problem is that the new rules for saturation in Listing 2.27 introduce new "possibilities" to find models of the program without the saturation predicate **ok**. To avoid this behavior, a rule like the one shown in Listing 2.30 is

introduced. The rule ensures that a model, which can only be induced by the additional rules in Listing 2.27, derives the saturation predicate **ok** and therefore again lead to saturation.

```
ok :- in(X),out(X).
```

Listing 2.30: rule to fix saturation

With this rule the solver at last yields the expected result-set `{s(a), s(b), in(a), in(b), out(a), out(b), ok}`.

2.6 Complexity

For this section we assume that the basic notions of complexity theory are familiar such as hardness, completeness and the classes P, NP, $\mathsf{co\text{-}NP}$. Further background can be found in [Joh90, Pap94].

2.6.1 Decision problems

To be able to talk about complexity, the specific problem to analyze has to be defined. To this end usually decision problems are specified. In general, decision problems are a set of problem instances which should be answered with yes resp. no. For our purpose we define two problems, namely *brave* and *cautious reasoning*:

Definition 2.33. *Brave reasoning (GRAPPA)*: Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance, σ a semantics and $s \in S$ a statement. Decide whether there exists a σ-interpretation v of S s. t. $v(s) = \mathbf{t}$?

Definition 2.34. *Cautious reasoning (GRAPPA)*: Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance, σ a semantics and $s \in S$ a statement. Decide whether $v(s) = \mathbf{t}$ holds for every σ-interpretation v of S.

Definition 2.35. *Brave reasoning (ASP)*: Let P be an ASP-program and a be a ground atom. Decide if there exists an answer-set of P in which a is included.

Definition 2.36. *Cautious reasoning (ASP)*: Let P be an ASP-program and a be a ground atom. Decide whether a is element of every answer-set of P.

2.6.2 The polynomial hierarchy

The *polynomial hierarchy* was first introduced in [SM73, Sto76]

Definition 2.37. The *polynomial hierarchy* is inductively defined as follows:

$$\Delta_0^{\mathsf{P}} := \Sigma_0^{\mathsf{P}} := \Pi_0^{\mathsf{P}} := \mathsf{P}$$
$$\Delta_{k+1}^{\mathsf{P}} := \mathsf{P}^{\Sigma_k^{\mathsf{P}}}$$

$$\Sigma^{\mathsf{P}}_{k+1} := \mathsf{NP}^{\Sigma^{\mathsf{P}}_k}$$
$$\Pi^{\mathsf{P}}_{k+1} := \text{co-}\Sigma^{\mathsf{P}}_{k+1}$$

For the cases $k = 1$ and $k = 2$ this yields:

$k = 1$:
$$\Delta^{\mathsf{P}}_1 = \mathsf{P}^{\Sigma^{\mathsf{P}}_0} = \mathsf{P}^{\mathsf{P}} = \mathsf{P}$$
$$\Sigma^{\mathsf{P}}_1 = \mathsf{NP}^{\Sigma^{\mathsf{P}}_0} = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$$
$$\Pi^{\mathsf{P}}_1 = \text{co-}\Sigma^{\mathsf{P}}_1 = \text{co-}\mathsf{NP}$$

$k = 2$:
$$\Delta^{\mathsf{P}}_2 = \mathsf{P}^{\Sigma^{\mathsf{P}}_1} = \mathsf{P}^{\mathsf{NP}}$$
$$\Sigma^{\mathsf{P}}_2 = \mathsf{NP}^{\Sigma^{\mathsf{P}}_1} = \mathsf{NP}^{\mathsf{NP}}$$
$$\Pi^{\mathsf{P}}_2 = \text{co-}\Sigma^{\mathsf{P}}_2 = \text{co-}\mathsf{NP}^{\mathsf{NP}}$$

The motivation for this hierarchy is based on the question: Suppose there is a problem A with a sub-problem B. Assume problem B can be solved with constant costs by a so-called *oracle*. What is the remaining complexity of problem A if the oracle is used to compute sub-problem B?

For example, assume a problem A is $\Pi^{\mathsf{P}}_2 = \text{co-}\mathsf{NP}^{\mathsf{NP}}$ complete. That means that A would be co-NP complete if there would be an oracle which is able to solve an NP hard sub-problem of A in constant time and space.

2.6.3 Complexity of GRAPPA

The results for GRAPPA presented in this section are actually results shown for ADFs but as argued in [BW14] these results can be carried over to GRAPPA.

	model	admissible	complete	preferred
brave	NP	Σ^{P}_2	Σ^{P}_2	Σ^{P}_2
cautious	co-NP	trivial	co-NP	Π^{P}_3

Table 2.1: complexity results for GRAPPA

In Table 2.1 complexity results for different semantics are given, which are all completeness results, and are proven for ADFs in [SW14].

2.6.4 Complexity of ASP-programs

The usual case for analyzing the complexity of an ASP-program is to consider the *data complexity* of the problem, i. e. the ASP-program is assumed to be static and only the input of the program is changing. The data complexity of a *normal* ASP-program for brave and cautious reasoning is NP resp. co-NP complete. However, for *disjunctive* ASP-programs the data complexity for brave and cautious reasoning increases, i. e. the problems are Σ^{P}_2 resp. Π^{P}_2 complete, which is a result of [EG95, EGM97].[10]

[10] An overview of the complexities for different ASP-fragments can be found in [LPF+06].

This is different for the *combined complexity* of ASP. In contrast to *data complexity*, *combined complexity* considers the case that the program is not longer static, i. e. not only the data can vary, but also the program. In general, the combined complexity of ASP is NEXP and $\mathsf{NEXP}^{\mathsf{NP}}$ for *normal* resp. *disjunctive* ASP-programs as shown in [EGM97, ELS98]. But this changes significantly if the program is restricted to predicates with *bounded arity*, i. e. there exists a constant $n \in \mathbb{N}$ such that the arity of every predicate occurring in the program is smaller than n. In this case brave and cautious reasoning for

- *normal* ASP-programs is Σ_2^{P} resp. Π_2^{P} complete
- *disjunctive* ASP-programs is Σ_3^{P} resp. Π_3^{P} complete

as shown in [EFFW07].

2.6.5 Consequences for ASP-encodings

The comparison of the complexity of GRAPPA and the data complexity of ASP indicates, that the preferred semantics can not be encoded in a *static* approach, because for cautious reasoning the complexity of the preferred semantics (Π_3^{P}) is higher than the data complexity of ASP (Π_2^{P}). Nonetheless it is possible to encode semantics with lower complexity this way, i. e. a fixed encoding can handle arbitrary GRAPPA-instances. Later we refer to this type of encoding as *static encoding*.

In contrary for combined complexity, the result that a disjunctive ASP-program with bounded arity is Σ_2^{P} resp. Π_3^{P} complete shows that an ASP-encoding for brave (Σ_2^{P}) resp. cautious (Π_3^{P}) reasoning for the preferred semantics can be found. But by definition of combined complexity, this encoding can not handle arbitrary GRAPPA-instances, but depends on a given GRAPPA-instance. Hence, the encoding must be generated for each GRAPPA-instance individually. Later we refer to this type of encoding as *dynamic encoding*.

CHAPTER 3

Static encodings

In this section the *static* encoding is presented to evaluate GRAPPA-instances under model, admissible or complete semantics. The name "static" was chosen because the encoding stays the same for every input instance (except the input itself).

The encoding itself is split into several parts, which correspond to separate files[1] , to make the encoding flexible to some extent.

- **Input:** Of course this part is always part of the program. The name of this file is arbitrary it only has to be added to the call of the ASP-solver. The input format is described in Section 3.1.
- **Basic definitions:** This part can be found in the file *basicdefs.lp* and includes the rules for basic evaluations of the input and is described in Section 3.3. This part is always part of the program.
- **Model semantics:** This part can be found in the file *model.lp* and includes the rules to calculate the model semantics on a given input. A detailed description of this part is given in Section 3.4. The definitions in this part are depending on *basicdefs.lp*.
- **Admissible semantics:** This part can be found in *admissible.lp* and includes the rules to calculate the admissible semantics on a given input. A detailed description of this part is given in Section 3.5. The definitions in this part are depending on *basicdefs.lp*.
- **Complete semantics:** This part can be found in *complete.lp* and includes the rules to calculate the admissible semantics on a given input. A detailed description of this part is given in Section 3.6. The definitions in this part are depending on *basicdefs.lp* and *admissible.lp*.

[1] The encoding can be found at `http://dbai.tuwien.ac.at/proj/adf/grappavis/`.

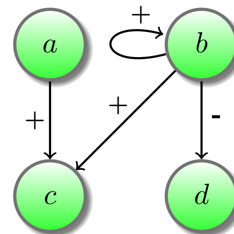
To evaluate a GRAPPA-instance under admissible or complete semantics, saturation is necessary. As described in Section 2.5.9 rules with aggregates must be handled with special care if saturation is used. Therefore, the rules in the basic definitions look quite complex, even for simple tasks, because the reformulations from Section 2.5.9 have been applied. Even though the model semantics does not need saturation, the same (saturation save) basic definitions are used for model semantics to avoid a second (equivalent) set of basic rules.

The listings which are presented in this chapter are directly imported from the original encodings. Therefore, the line-numbers of these listings match the actual line-numbers of the corresponding file.

3.1 Description of the input

In this section the system to encode a GRAPPA-instance for the static encoding is described. First an example of a small instance is given to get an idea how the input looks like. Then the different predicates, which are used to specify the instance, are described in detail.

Example 3.1. (the same instance as in Example 2.4) Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance with $S = \{a, b, c, d\}$ and $L = \{+, \text{-}\}$. The graph on the right side shows the labels of each link. The acceptance condition for all nodes is



$$\#_t(+) - \#(+) = 0 \wedge \#(\text{-}) = 0.$$

G is represented by the encoding shown in Listing 3.1. For every edge in E an atom **e** with three parameters is inserted. The first argument is the source of the edge, the second parameter is the target of the edge. The third parameter states the label of the edge. For every node in S an atom **s** with two parameters is inserted. The first parameter is the label of the node and the second parameter specifies the acceptance condition.

The smallest entity of an acceptance-condition is a single comparison which is captured by the atom **basicpattern**. If an acceptance-condition consists of several comparisons – i. e. **basicpattern** – which are linked by boolean operators (and, or, xor) the **basicpattern**s are arguments of corresponding predicates **and**, **or**, **xor**.

A **basicpattern** represents a comparison, therefore it takes three arguments, namely the left-hand-side (LHS), the comparator and the right-hand-side (RHS). The comparator is just an element of the set {`"<"`, `"<="`, `"!="`, `"=="`, `"=>"`, `">"`}. The RHS is an integer and the LHS is a unique id which references a sum of terms which is referred to as functional combination.

That means that for every id within a LHS of a **basicpattern** one or more **term** atoms are inserted. One **term** atom represents a single summand of a functional combination

and takes four arguments. The first argument is the id of the functional combination the term belongs to, the second argument is a unique id of the single terms within one functional combination. The third argument is the coefficient which is multiplied to the result of the function given in the fourth argument.

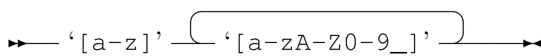
```
1 minDom(-5).
2 maxDom(5).
3
4 s(a,and(basicpattern(a_0,"==",0),basicpattern(a_1,"==",0))).
5 s(c,and(basicpattern(a_2,"==",0),basicpattern(a_3,"==",0))).
6 s(b,and(basicpattern(a_4,"==",0),basicpattern(a_5,"==",0))).
7 s(d,and(basicpattern(a_6,"==",0),basicpattern(a_7,"==",0))).
8
9 e(a,c,"+").
10 e(b,c,"+").
11 e(b,d,"-").
12 e(b,b,"+").
13
14 term(a_0,0,1,activelabelcount(a,"+")).
15 term(a_0,1,-1,labelcount(a,"+")).
16
17 term(a_1,0,1,activelabelcount(a,"-")).
18
19 term(a_2,0,1,activelabelcount(c,"+")).
20 term(a_2,1,-1,labelcount(c,"+")).
21
22 term(a_3,0,1,activelabelcount(c,"-")).
23
24 term(a_4,0,1,activelabelcount(b,"+")).
25 term(a_4,1,-1,labelcount(b,"+")).
26
27 term(a_5,0,1,activelabelcount(b,"-")).
28
29 term(a_6,0,1,activelabelcount(d,"+")).
30 term(a_6,1,-1,labelcount(d,"+")).
31
32 term(a_7,0,1,activelabelcount(d,"-")).
```

Listing 3.1: input example △

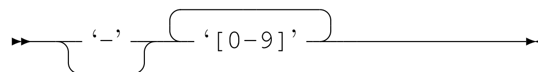
3.1.1 Syntax

Here the syntax diagrams of the more complicated constructs are given.

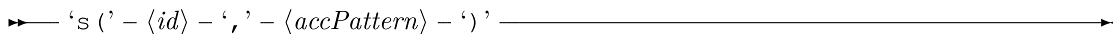
id:



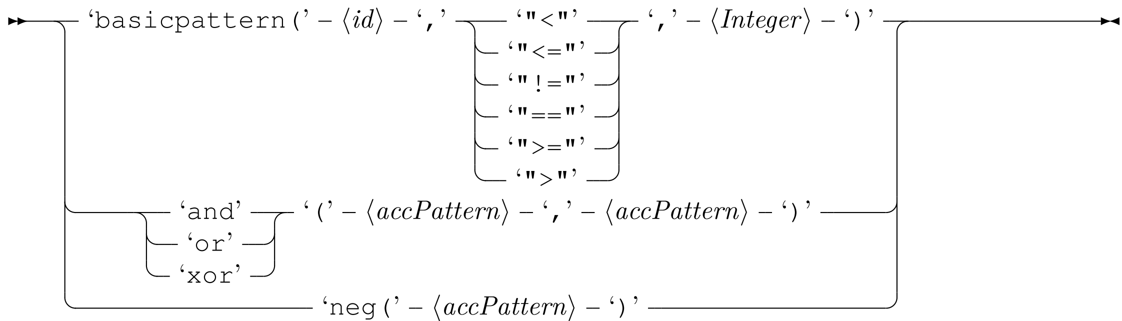
Integer:



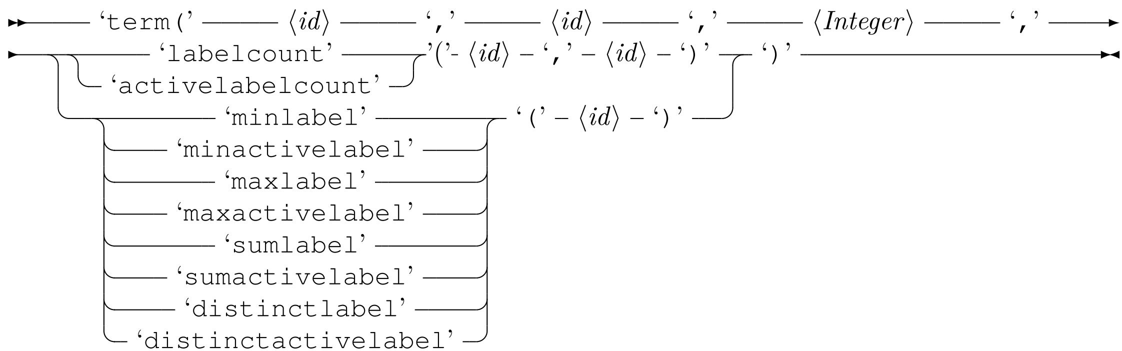
s:



accPattern:



term:



3.1.2 List of all predicates

The predicates are listed in alphabetical order.

activelabelcount(`node`, `edgelabel`)

description: Corresponds to the function #(*edgelabel*) from the GRAPPA syntax. Instructs the program to count all active edges with the label given by `edgelabel` on the given `node`. The edgelabel is stated without quotes if numeric, otherwise with quotes.
e. g. activelabelcount(node1, 124). activelabelcount(node2, "+").

{and | or | xor}(`LHS`, `RHS`)

`LHS` Left hand side of the operator.

`RHS` Right hand side of the operator.

description: Represents the correspondent boolean function. `LHS` resp. `RHS` must be an atom built by one of the following predicates: **basicpattern**, **and**, **or**, **neg** or **xor**.

basicpattern(`sumId`, `comparator`, `RHS`)

`sumId` The id of a functional combination on the left hand side of the comparator.

`comparator` One of the following operators (inclusive the quotation marks): "<=", "<", "==", "!=", ">", ">="

`RHS` The right hand side of the operator which must be an integer value.

description: This predicate defines a comparison of the functional combination – on the LHS defined by `sumId` – and the integer given by `RHS`. The functional combination is represented by a set of **term** predicates which share the same id.
e. g. basicpattern(a1, "==", 13)

distinctactivelabel(`node`)

description: Corresponds to the function *count*() from the GRAPPA syntax. Instructs the program to retrieve the number of active edges with different labels on the given `node`.

distinctlabel(`node`)

description: Corresponds to the function $count_t()$ from the GRAPPA syntax. Instructs the program to retrieve the number of edges – active or not – with different labels on the given `node`.

e(`source`, `target`, `label`)

`source` The `name` of the source-node.

`target` The `name` of the target-node.

`label` The edgelabel. If numeric without, otherwise with quotes.

description: Defines a (directed) edge between source and target with a given label.
e. g. e(n1, n2, 23). e(a, b, "+").

labelcount(`node`, `edgelabel`)

description: Corresponds to the function $\#_t(edgelabel)$ from the GRAPPA syntax. Instructs the program to count all edges – active or not – with the label given by `edgelabel` on the given `node`.
e. g. labelcount(a, -12). labelcount(b, "attack").

maxactivelabel(`node`)

description: Corresponds to the function $max()$ from the GRAPPA syntax. Instructs the program to retrieve the maximum edgeweight of all active edges on the given `node`. This is undefined if there are edges with non numeric values.

maxDom(`maxValue`)

description: The min-aggregate yields ∞ in case of an empty set. To avoid this case `maxValue` is added to the set under evaluation as fallback-value. To improve the performance `maxValue` should be as low as possible.

maxlabel(`node`)

description: Corresponds to the function $max_t()$ from the GRAPPA syntax. Instructs the program to retrieve the maximum edgeweight of all edges – active or not – on the given `node`. This is undefined if there are edges with non numeric values.

minactivelabel(`node`)

description: Corresponds to the function $min()$ from the GRAPPA syntax. Instructs the program to retrieve the minimum edgeweight of all active edges on the given `node`. This is undefined if there are edges with non numeric values.

minDom(`minValue`)

description: The max-aggregate yields $-\infty$ in case of an empty set. To avoid this case `minValue` is added to the set under evaluation as fallback-value. To improve the performance `minValue` should be as high as possible.

minlabel(`node`)

description: Corresponds to the function $min_t()$ from the GRAPPA syntax. Instructs the program to retrieve the minimum edgeweight of all edges – active or not – on the given `node`. This is undefined if there are edges with non numeric values.

neg(`arg`)

`arg` The argument of the negation.

description: Represents the negation. `arg` must be an atom built by one of the following predicates: **basicpattern**, **and**, **or**, **neg** or **xor**.

s(`name`, `pattern`)

`name` The name of the node.

`pattern` The acceptance condition of the node.

description: This predicate defines a node with its name and acceptance-condition. The condition must be provided in `pattern` and must be constructed by of the following predicates: **basicpattern**, **and**, **or**, **neg** or **xor**.

sumactivelabel(`node`)

description: Corresponds to the function *sum*() from the GRAPPA syntax. Instructs the program to retrieve the sum of all edgeweights of all active edges on the given `node`. This is undefined if there are edges with non numeric values.

sumlabel(`node`)

description: Corresponds to $sum_t()$ from the GRAPPA syntax. Instructs the program to retrieve the sum of all edgeweights of all edges – active or not – on the given `node`. This is undefined if there are edges with non numeric values.

term(`sumId`, `id`, `factor`, `function`)

`sumId` The id for the functional combination this term belongs to.

`id` A unique id of this **term** within all terms of the functional combination referenced by `sumId`.

`factor` The coefficient (integer) which is multiplied to the result of `function`

`function` One of the following predicates: **activelabelcount**, **labelcount**, **minactivelabel**, **minlabel**, **maxactivelabel**, **maxlabel**, **sumactivelabel**, **sumlabel**, **distinctactivelabel**, **distinctlabel**

description: A single **term** predicate represents a term of the form $a \cdot f(\cdot)$ where a is an integer given by `factor` and $f(\cdot)$ is the function given by `function`. One or more **term** predicates together are used to represent functional combinations which are necessary for **basicpattern**-predicates. `sumId` defines the functional combination the **term** belongs to. The additional `id` is necessary to distinguish duplicate expressions within a functional combination.

3.1.3 References to the definitions

The rules for deriving the different predicates are defined in different listings. To provide a fast way to look up the definitions, an index is given in Table 3.1, where for each predicate the listings are given in which the predicate is derived.

predicate	defined in	predicate	defined in
accept	3.8, 3.28	maxalabel	3.18, 3.28
activeedge	3.14, 3.28	maxalabel_dom	3.18
alcount	3.15, 3.28	maxtlabel	3.11
alcount_dom	3.15	minalabel	3.17, 3.28
alcount_max	3.15	minalabel_dom	3.17
basicpattern	3.4	mterm	3.22, 3.28
cntActLabel	3.17, 3.18	mterm_max	3.22
cntActLabel_dom	3.17, 3.18	nomodel	3.7, 3.28
distactivelabel	3.20, 3.28	notaccept	3.8, 3.28
distactivelabel_dom	3.20	mintlabel	3.10
distlabel	3.13	ok	3.27, 3.28
false	3.6, 3.28	out	3.24, 3.25, 3.26
guess	3.24, 3.26, 3.29	pattern	3.3
in	3.24, 3.25, 3.26	sumalabel	3.19, 3.28
ismodel	3.7, 3.28	sumalabel_dom	3.19
label_max	3.21	sumalabel_max	3.19
label_min	3.21	sumalabel_min	3.19
lcount	3.9	sumtlabel	3.12
lhspat	3.23, 3.28	true	3.5, 3.28
lhspat_dom	3.23	undec	3.25

Table 3.1: index for the definition of the predicates

3.2 States of nodes

When a GRAPPA-instance is evaluated under some semantics, a state is assigned to every node. These states are represented by the following predicates in the encoding. The argument INST introduced in order to use "subprograms" as explained in Section 2.5.7.

in(INST, NODE)

description: This state corresponds to the fact that NODE is accepted.

out(INST, NODE)

description: This state corresponds to the fact that NODE is not accepted.

undec(INST, NODE)

description: Marks NODE as undecided.

3.3 Basic definitions

This part contains basic rules which are used by the encodings of all semantics.

3.3.1 Basic facts

First there are some facts in Listing 3.2 which define the syntax of the relational operators. Moreover, a large number is required later which is defined in Line 243.

```
237 eq("==").
238 neq("!=").
239 leq("<=").
240 geq(">=").
241 gt(">").
242 lt("<").
243 maxVal(1661992959).
```

Listing 3.2: definition of relational operators

3.3.2 Parsing the input

The acceptance condition – as it is represented via the predicate **s** – is broken down to its single components which are the nested **and**, **or**, **xor**, **neg** predicates and finally the **basicpattern**. This is done in a recursive fashion as shown in Listing 3.3. The rules traverse the syntax-tree of the acceptance-condition given in **s** and "wraps" every boolean operation, as well as its arguments, in a new **pattern** predicate.

```
228 pattern(P) :- s(X,P).
229 pattern(P) :- pattern(and(P,_)).
230 pattern(P) :- pattern(and(_,P)).
231 pattern(P) :- pattern(or(P,_)).
232 pattern(P) :- pattern(or(_,P)).
233 pattern(P) :- pattern(neg(P)).
234 pattern(P) :- pattern(xor(P,_)).
235 pattern(P) :- pattern(xor(_,P)).
```

Listing 3.3: parsing patterns

The predicate **basicpattern** is required in some of the rules defined later. To provide "direct access" the rule shown in Listing 3.4 is added.

```
226 basicpattern(L,R,A) :- pattern(basicpattern(L,R,A)).
```

Listing 3.4: retrieve **basicpattern**

3.3.3 Retrieving the truth values

Now the truth values of all **pattern** atoms must be computed. The start is done with the **basicpattern** atoms, then the result is propagated through possible boolean functions. In a first step true resp. false **basicpattern** predicates are identified. Recall that a **basicpattern** represents a functional combination whose result is compared to an integer value. So the value of the functional combination has to be calculated before this result can be compared to the integer value. Assume at this point that this is already done and the result is stored in the predicate **lhspat**.

lhspat(`INST`, `L`, `I`)

- `INST` Argument used as described in Section 2.5.7
- `L` The id of the functional combination.
- `I` The result of the functional combination.

The rules for **true** resp. **false** are given in the Listing 3.5 resp. 3.6. The rules mainly check what kind of comparison is in use. They extract the operator – encoded as string – from the **basicpattern** and store it in the variable `R`. In each rule the content of `R` is matched to one of the **geq**, **leq**, **eq**, **neq**, **gt**, **lt** atoms and the corresponding comparison is done.

The argument `INST` which appears in almost every rule is introduced as a consequence of the methodology described in Section 2.5.7.

```
189 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), geq(R), I>=A,
    ↪ basicpattern(L,R,A).
190 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), leq(R), I<=A,
    ↪ basicpattern(L,R,A).
191 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), eq(R), I=A,
    ↪ basicpattern(L,R,A).
192 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), neq(R), I!=A,
    ↪ basicpattern(L,R,A).
193 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), gt(R), I>A,
    ↪ basicpattern(L,R,A).
194 true(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), lt(R), I<A,
    ↪ basicpattern(L,R,A).
```

Listing 3.5: identify true **basicpattern**

The rules for the **false** predicate in Listing 3.6 are similar to the rules of the **true** predicate. The only difference is that the inverse operation – to the operation stored in variable `R` – is carried out to retrieve the result.

```
196 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), geq(R), I<A,
    ↪ basicpattern(L,R,A).
197 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), leq(R), I>A,
    ↪ basicpattern(L,R,A).
```

```
198 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), eq(R), I!=A,
      ↪ basicpattern(L,R,A).
199 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), neq(R), I=A,
      ↪ basicpattern(L,R,A).
200 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), gt(R), I<=A,
      ↪ basicpattern(L,R,A).
201 false(INST,X) :- X=basicpattern(L,R,A), lhspat(INST,L,I), lt(R), I>=A,
      ↪ basicpattern(L,R,A).
```

Listing 3.6: identify false **basicpattern**

The truth values of the **basicpattern** atoms are used to compute the truth values of the **pattern** atoms. An atom **ismodel** or **nomodel** is derived for a specific **pattern** atom if the **pattern** is true resp. false.

The base case is **pattern**(X) where X is a **basicpattern** which is represented by the rules in Line 207 resp. 215 in Listing 3.7. The other rules cover the cases where **pattern** is a boolean combination of **basicpattern** atoms. The rules corresponding to a boolean operation fire when the atoms **nomodel**, **ismodel** – corresponding to the boolean operation to be evaluated – have been derived.

```
207 ismodel(INST,X) :- pattern(X), true(INST,X).
208 ismodel(INST,X) :- pattern(X), pattern(X1), X=neg(X1), nomodel(INST,X1).
209 ismodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=and(X1,X2),
      ↪ ismodel(INST,X1), ismodel(INST,X2).
210 ismodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=or(X1,X2),
      ↪ ismodel(INST,X1).
211 ismodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=or(X1,X2),
      ↪ ismodel(INST,X2).
212 ismodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=xor(X1,X2),
      ↪ ismodel(INST,X1), nomodel(INST,X2).
213 ismodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=xor(X1,X2),
      ↪ ismodel(INST,X2), nomodel(INST,X1).
214
215 nomodel(INST,X) :- pattern(X), false(INST,X).
216 nomodel(INST,X) :- pattern(X), pattern(X1), X=neg(X1), ismodel(INST,X1).
217 nomodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=and(X1,X2),
      ↪ nomodel(INST,X1).
218 nomodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=and(X1,X2),
      ↪ nomodel(INST,X2).
219 nomodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=or(X1,X2),
      ↪ nomodel(INST,X1), nomodel(INST,X2).
220 nomodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=xor(X1,X2),
      ↪ nomodel(INST,X1), nomodel(INST,X2).
221 nomodel(INST,X) :- pattern(X), pattern(X1), pattern(X2), X=xor(X1,X2),
      ↪ ismodel(INST,X1), ismodel(INST,X2).
```

Listing 3.7: calculate the truth value of acceptance patterns

Finally with the information held by **nomodel**, **ismodel** the program can decide whether a node is accepted or not. This is achieved by the rules in Listing 3.8 and the information is stored in the atoms **accept** resp. **notaccept**.

```
203 accept(INST,X) :- s(X,I), ismodel(INST,I).
204 notaccept(INST,X) :- s(X,I), nomodel(INST,I).
```

Listing 3.8: acceptance of a node **basicpattern**

3.3.4 Calculate functions unaffected by active edges

Still the functional combinations have to be evaluated and, hence, the result of the functions appearing in the functional combinations need to be calculated. In this subsection the rules of the functions, which do not depend on active edges – therefore not affected by saturation – are presented in the Listings 3.9 to 3.13. The definitions are straight forward implemented by making use of the corresponding ASP-aggregate functions.

Regarding **lcount** – presented in Listing 3.9 – a potential pitfall would have been to count the edgelabels itself, because two edges could have the same label and then they would be counted as one. Hence, the source nodes are counted, because they must be unique and multiedges are not allowed by definition.

```
27 lcount(NODE,L,NUMBER) :- s(NODE,_), term(_,_,_,labelcount(NODE,L)),
    ↪ NUMBER = #count{I:e(I,NODE,L)}.
```

Listing 3.9: rule definition for **lcount**

```
52 mintlabel(NODE,NUMBER) :- s(NODE,_), term(_,_,_,minlabel(NODE)),
    ↪ NUMBER= #min{I:e(_,NODE,I);Max:maxDom(Max)}.
```

Listing 3.10: rule definition for **mintlabel**

```
77 maxtlabel(NODE,NUMBER) :- s(NODE,_), term(_,_,_,maxlabel(NODE)),
    ↪ NUMBER= #max{I:e(_,NODE,I);Min:minDom(Min)}.
```

Listing 3.11: rule definition for **maxtlabel**

Regarding **sumtlabel** presented in Listing 3.12: the elements of the set, which the ASP-aggregate is evaluating, are tuples and not single elements. Otherwise different edges with the same weight would contribute only once to the sum. To distinguish such values, the elements are tuples consisting of the edgeweight and the source node.

```
97 sumtlabel(NODE,NUMBER) :- s(NODE,_), term(_,_,_,sumlabel(NODE)),
    ↪ NUMBER= #sum{I,S:e(S,NODE,I)}.
```

Listing 3.12: rule definition for **sumtlabel**

Finally, the reason why **distlabel** in Listing 3.13 is also calculated for **distintactivelabel** is that this value is required by the definition of **distactivelabel** which is presented later.

```
116 distlabel(NODE,NUMBER) :- s(NODE,_), term(_,_,_,distinctactivelabel(NODE)),
    ↪ NUMBER=#count{I:e(_,NODE,I)}.
117 distlabel(NODE,NUMBER) :- s(NODE,_), term(_,_,_,distinctlabel(NODE)),
    ↪ NUMBER=#count{I:e(_,NODE,I)}.
```

Listing 3.13: rule definition for **distlabel**

3.3.5 Calculate functions affected by active edges

In this subsection the rules for the functions are presented which have to deal with active edges. To retrieve the information if an edge is active the predicate **activeedge** is used. An active edge is an edge where its source node is accepted and this definition is covered by the rule in Listing 3.14.

```
8 activeedge(INST,Source,Target,Label) :- e(Source,Target,Label),
    ↪ in(INST,Source).
```

Listing 3.14: rule definition for **activeedge**

As described in Section 2.5.9 aggregates computed over sets of atoms affected by saturation are requiring special care. These considerations are applied and yield the rules given in Listing 3.15 which define the predicate **alcount**. In Line 14 the maximum number of active edges – for a given label – is calculated for each node where **activelabelcount** is applied to. This way the domain for this predicate can be defined – in Line 15 – as it simply ranges from zero to the maximum. The rule itself is just a reformulation of a `#count` aggregate as shown in Listing 2.22. Only the predicate **guess** is added which is explained in Section 2.5.7. When and how the predicate **ok** is derived to "trigger" the saturation process is shown in the Sections 3.5 and 3.6.

```
14 alcount_max(NODE, L, MAX) :- s(NODE,_),term(_,_,_,activelabelcount(NODE,L)),
    ↪ MAX = #count{I:e(I, NODE, L)}.
15 alcount_dom(NODE,L,0..MAX) :- alcount_max(NODE, L, MAX).
16
17 alcount(INST,NODE,L,NUMBER) :- alcount_dom(NODE,L,NUMBER), guess(INST),
18                   NUMBER = #sum{
19                        1, I:activeedge(INST,I,NODE,L);
20                       -1, I:activeedge(INST,I,NODE,L), ok(INST);
21                       NUMBER: ok(INST)
22                       }.
```

Listing 3.15: rule definitions for **activelabelcount**

For the rules defining **minactivelabel** the predicate **minalabel** is used to store the minimum of all active edges. A simple approach is presented in Listing 3.16.

```
minalabel(NODE,L,NUMBER) :-
               NUMBER = #min{L:activeedge(INST,I,NODE,L); Max: maxDom(Max)}.
```

Listing 3.16: a straight forward rule for **minalabel**

Within the aggregate the entry "`Max: maxDom(Max)`" maybe surprising but the addition is necessary to cover the case when there are no active edges at all. Without this element the set would be empty and the aggregate would return ∞ – the neutral element of the `min` function. For obvious practical reasons this case is avoided by adding the upper bound of the domain.

Again the reformulation from Section 2.5.9 regarding saturation have to be applied. With the considerations from Listing 2.23 the quite simple rule from Listing 3.16 becomes the program in Listing 3.17.

```
32 minalabel_dom(NODE, Max) :-  s(NODE,_), term(_,_,_,minactivelabel(NODE)),
    ↪ maxDom(Max).
33 minalabel_dom(NODE, NUMBER) :-  s(NODE,_), term(_,_,_,minactivelabel(NODE)),
    ↪ e(_,NODE,NUMBER).
34
35 cntActLabel_dom(NODE,0..NUMBER) :-  minalabel_dom(NODE,_), NUMBER =
    ↪ #count{S: e(S,NODE,I)}.
36 cntActLabel(INST,NODE,NUMBER+1) :- cntActLabel_dom(NODE,NUMBER),
    ↪ guess(INST),
37                   NUMBER = #sum{ 1,S: activeedge(INST,S,NODE,I);
38                           -1,S: activeedge(INST,S,NODE,I),ok(INST);
39                            NUMBER:ok(INST)
40                           }.
41
42 minalabel(INST,NODE,NUMBER) :- minalabel_dom(NODE,NUMBER), guess(INST),
    ↪ cntActLabel(INST,NODE,N),
43                   0 < #sum{
44                       1-N*(NUMBER-I):activeedge(INST,_,NODE,I),I<=NUMBER;
45                       1-N*(NUMBER-Max): maxDom(Max), Max<=NUMBER;
46                       Z:maxVal(Z),ok(INST)
47                       }.
```

Listing 3.17: rule definition for **minalabel**

The argument "`NUMBER+1`" of **cntActLabel** – in Line 36 of Listing 3.17 – includes "`+1`" because the upper bound of the domain is also part of the set as argued before. To ensure that the sum becomes greater than zero – in case of saturation – the atom **maxVal** is used in Line 46.

The domain of **minalabel** – represented by **minalablel_dom** – includes all labels of the incoming edges of the node – collected in Line 33 – and additionally the upper bound of the domain – added in Line 32. The domain of **cntActLabel** is represented by **dom_cntActLabel** and ranges from zero to the number of incoming edges for the current node – defined in Line 35.

For **maxactivelabel** the predicate **maxalabel** is used to store the maximum of all active edges. The rules – given in Listing 3.18 – are retrieved in the same manner as for **minactivelabel** with the reformulation of `#max` instead of `#min`. To exclude $-\infty$ as result of the `#max`-aggregate – in case of an empty set – the lower bound of the domain is added to the set under evaluation.

```
57 maxalabel_dom(NODE, Min) :-  s(NODE,_), term(_,_,_,maxactivelabel(NODE)),
     ↪ minDom(Min).
58 maxalabel_dom(NODE, NUMBER) :-  s(NODE,_), term(_,_,_,maxactivelabel(NODE)),
     ↪ e(_,NODE,NUMBER).
59
60 cntActLabel_dom(NODE,0..NUMBER) :-  maxalabel_dom(NODE,_), NUMBER =
     ↪ #count{S: e(S,NODE,I)}.
61 cntActLabel(INST,NODE,NUMBER+1) :- cntActLabel_dom(NODE,NUMBER),
     ↪ guess(INST),
62                  NUMBER = #sum{ 1,S: activeedge(INST,S,NODE,I);
63                          -1,S: activeedge(INST,S,NODE,I),ok(INST);
64                           NUMBER:ok(INST)
65                          }.
66
67 maxalabel(INST,NODE,NUMBER) :- maxalabel_dom(NODE,NUMBER), guess(INST),
     ↪ cntActLabel(INST,NODE,N),
68                  0 < #sum{
69                      1+N*(NUMBER-I):activeedge(INST,_,NODE,I),-I<=-NUMBER;
70                      1+N*(NUMBER-Min): minDom(Min), -Min <=-NUMBER;
71                      Z:maxVal(Z),ok(INST)
72                      }.
```

Listing 3.18: rule definition for **maxalabel**

For **sumactivelabel** – defined in Listing 3.19 – the predicate **sumalabel** is used to store the sum of all active edges. Here the reformulation of Listing 2.21 is applied. To retrieve the upper bound of the domain all positive values are summed up, because there is no possibility to get a bigger result. In analogous fashion, for the lower bound all negative values are summed up.

```
82 sumalabel_max(NODE,Max) :- s(NODE,_), term(_,_,_,sumactivelabel(NODE)), Max=
     ↪ #sum{I,S:e(S,NODE,I), I>0}.
83 sumalabel_min(NODE,Min) :- s(NODE,_), term(_,_,_,sumactivelabel(NODE)), Min=
     ↪ #sum{I,S:e(S,NODE,I), I<0}.
84
85 sumalabel_dom(NODE,Min..Max) :- sumalabel_max(NODE,Max), sumalabel_min(NODE,
     ↪ Min).
86
87 sumalabel(INST,NODE,NUMBER) :- sumalabel_dom(NODE, NUMBER), guess(INST),
88                NUMBER= #sum{
89                   I,S:activeedge(INST,S,NODE,I);
90                  -I,S:activeedge(INST,S,NODE,I), ok(INST);
91                  NUMBER: ok(INST)
92                  }.
```

Listing 3.19: rule definitions for **sumalabel**

For **distactivelabel** the predicate **distinctactivelabel** – presented in Listing 3.20 – is used to store the number of active edges having different labels. Here the use of a #count-aggregate would be the natural choice, same as for **alcount**. Therefore, the reformulation of Listing 2.22 is required again. It is easy to see that the domain ranges

from zero to the number of edges with different labels which corresponds precisely to the definition of the predicate **distlabel**.

```
102 distactivelabel_dom(NODE, 0..NUMBER) :- distlabel(NODE,NUMBER).
103
104 distactivelabel(INST,NODE,NUMBER):- s(NODE,_),
    ↪ term(_,_,_,distinctactivelabel(NODE)),
105                 distactivelabel_dom(NODE, NUMBER), guess(INST),
106                   NUMBER=#sum{
107                        1,I:activeedge(INST,_,NODE,I);
108                       -1,I:activeedge(INST,_,NODE,I), ok(INST);
109                       NUMBER,ok: ok(INST)
110                       }.
```

Listing 3.20: rule definition for **distactivelabel**

3.3.6 Calculate functional combinations

With the function values now being available the coefficients have to be incorporated to finally calculate the result of every single term. To this end the predicate **mterm** – defined in Listing 3.22 – is used. The predicate **mterm_max** – also defined in this listing – stores the minimum and maximum of possible values for a single term. This is necessary because rules – defined later – require the domain for each term.

For the domain of **minalabel** and **maxalabel** the minimum and maximum weight of incoming edges is required. This information is retrieved by the rules given in Listing 3.21 and stored by the predicates **label_max** resp. **label_min**.

```
128 label_max(NODE, MAX) :- s(NODE,_), term(_,_,_,minactivelabel(NODE)), MAX=
    ↪ #max{I:e(_,NODE,I)}.
129 label_max(NODE, MAX) :- s(NODE,_), term(_,_,_,maxactivelabel(NODE)), MAX=
    ↪ #max{I:e(_,NODE,I)}.
130 label_min(NODE, MIN) :- s(NODE,_), term(_,_,_,minactivelabel(NODE)), MIN=
    ↪ #min{I:e(_,NODE,I)}.
131 label_min(NODE, MIN) :- s(NODE,_), term(_,_,_,maxactivelabel(NODE)), MIN=
    ↪ #min{I:e(_,NODE,I)}.
```

Listing 3.21: retrieve min/ max edgelabels

The definitions for **mterm** – in Listing 3.22 – for the different aggregate functions of GRAPPA– e. g. **activelabelcount** in Line 137 – are straight forward. The result is calculated by fetching the result of the involved function from the corresponding predicate – for **activelabelcount** the predicate **alcount** – and multiplying it with the coefficient of the term.

Also the definitions of **label_max**, for functions not depending on active edges, are simple, because the domain of these functions consists only of one value, namely the same value which is calculated in the corresponding **mterm**. However, the other definitions of **label_max** – depending on active edges – require further consideration.

The first approach to define **mterm_max** would be to set the minimum to zero and the maximum to the result of **alcount_max**. But the coefficient could also be negative which would cause the limits to switch, i. e. the minimum would be the result of **alcount_max** multiplied by the coefficient and the maximum zero. To handle both cases – positive and negative coefficient – the bounds are determined by min/ max aggregates on the set `{M * NUMBER;0}` which yield the correct bounds in either case. This consideration leads to the definition of **mterm_max** for the predicate **activelabelcount** in Line 136 of Listing 3.22.

The situation is similar for the definition of **mterm_max** – in Line 142 – for the predicate **minactivelabel**. The first idea would be to set the lower bound to the lowest edgeweight of the incoming edges multiplied with the coefficient and the maximum to the highest weight multiplied with the coefficient. The first problem of this approach arises if there are no incoming edges which would cause the result of **minalabel** to be the value given by **maxDom**. Therefore, the upper bound of **mterm** is set to **maxDom** multiplied with the coefficient. To handle negative coefficients, the bounds are selected again by min/ max aggregates. The definition of **mterm_max** for **maxactivelabel** in Line 148 follows the same approach.

The definition of **mterm_max** for the predicate **sumactivelabel** – as shown in Line 154 – is straight forward. Zero is always the lower/ upper bound – in case there are no incoming edges. The handling of negative coefficients is the same as before for **alcount_max**.

Finally, the definition of **mterm_max** for the predicate **distinctactivelabel** in Line 160 is the same as for the predicate **activelabelcount**.

```
136 mterm_max(X, ID, Y, Z) :- term(X,ID,M,I), I=activelabelcount(J,K),
      ↪ alcount_max(J,K,NUMBER), Y=#min{M*NUMBER;0}, Z=#max{M*NUMBER;0}.
137 mterm(INST,X, ID, Y)        :- term(X,ID,M,I), I=activelabelcount(J,K),
      ↪ alcount(INST,J,K,NUMBER), Y=M*NUMBER.
138
139 mterm_max(X, ID, Y, Y) :- term(X,ID,M,I), I=labelcount(J,K),
      ↪ lcount(J,K,NUMBER), Y=M*NUMBER.
140 mterm(INST,X, ID, Y)        :- term(X,ID,M,I), I=labelcount(J,K),
      ↪ lcount(J,K,NUMBER), Y=M*NUMBER, guess(INST).
141
142 mterm_max(X, ID, Y, Z) :- term(X,ID,M,I), I=minactivelabel(J), maxDom(Max),
      ↪ label_min(J,Min), Y=#min{M*Min; M*Max}, Z=#max{M*Min; M*Max}.
143 mterm(INST,X, ID, Y)        :- term(X,ID,M,I), I=minactivelabel(J),
      ↪ minalabel(INST,J,NUMBER), Y=M*NUMBER.
144
145 mterm_max(X, ID, Y, Y) :- term(X,ID,M,I), I=minlabel(J),
      ↪ mintlabel(J,NUMBER), Y=M*NUMBER.
146 mterm(INST,X, ID, Y)        :- term(X,ID,M,I), I=minlabel(J),
      ↪ mintlabel(J,NUMBER), Y=M*NUMBER, guess(INST).
147
148 mterm_max(X, ID, Y, Z) :- term(X,ID,M,I), I=maxactivelabel(J),
      ↪ label_max(J,Max), minDom(Min), Y=#min{M*Min;M*Max},
      ↪ Z=#max{M*Min;M*Max}.
149 mterm(INST,X, ID, Y)        :- term(X,ID,M,I), I=maxactivelabel(J),
```

```
    ↪ maxalabel(INST,J,NUMBER), Y=M*NUMBER, NUMBER<>#inf.
150
151 mterm_max(X, ID, Y, Y) :- term(X,ID,M,I), I=maxlabel(J),
    ↪ maxtlabel(J,NUMBER), Y=M*NUMBER.
152 mterm(INST,X, ID, Y)       :- term(X,ID,M,I), I=maxlabel(J),
    ↪ maxtlabel(J,NUMBER), Y=M*NUMBER, guess(INST).
153
154 mterm_max(X, ID, Y, Z) :- term(X,ID,M,I), I=sumactivelabel(J),
    ↪ sumalabel_min(J,Min), sumalabel_max(J,Max), Y=#min{M*Min;M*Max;0},
    ↪ Z=#max{M*Min;M*Max;0}.
155 mterm(INST,X, ID, Y)       :- term(X,ID,M,I), I=sumactivelabel(J),
    ↪ sumalabel(INST,J,NUMBER), Y=M*NUMBER.
156
157 mterm_max(X, ID, Y, Y) :- term(X,ID,M,I), I=sumlabel(J),
    ↪ sumtlabel(J,NUMBER), Y=M*NUMBER.
158 mterm(INST,X, ID, Y)       :- term(X,ID,M,I), I=sumlabel(J),
    ↪ sumtlabel(J,NUMBER), Y=M*NUMBER, guess(INST).
159
160 mterm_max(X, ID, Y, Z) :- term(X,ID,M,I), I=distinctactivelabel(J),
    ↪ distlabel(J,NUMBER), Y=#min{M*NUMBER;0},  Z=#max{M*NUMBER;0}.
161 mterm(INST,X, ID, Y)       :- term(X,ID,M,I), I=distinctactivelabel(J),
    ↪ distactivelabel(INST,J,NUMBER), Y=M*NUMBER.
162
163 mterm_max(X, ID, Y, Y) :- term(X,ID,M,I), I=distinctlabel(J),
    ↪ distlabel(J,NUMBER), Y=M*NUMBER.
164 mterm(INST,X, ID, Y)       :- term(X,ID,M,I), I=distinctlabel(J),
    ↪ distlabel(J,NUMBER), Y=M*NUMBER, guess(INST).
```

Listing 3.22: rule definition for **mterm** and **mterm_max**

In order to compute the truth values of the acceptance conditions of GRAPPA-instance, the predicate **lhspat** is defined in Listing 3.23 to sum up all terms of a functional combination. Without saturation it would be a "normal" sum aggregate, but due to saturation the aggregate is again reformulated as shown in Listing 2.21. For this reformulation a domain predicate **lhspat_dom** is necessary, which can easily be computed by summing up the lower resp. upper bound of every term involved. The lower and upper bound is already computed for every term and stored within **mterm_max** atoms.

```
172 lhspat_dom(X, MIN .. MAX) :-   basicpattern(X,_,_),
173           MIN = #sum{I, ID: mterm_max(X,ID,I,_)},
174           MAX = #sum{I, ID: mterm_max(X,ID,_,I)}.
175
176 lhspat(INST,X,Y) :-   Y = #sum {
177                  I, ID: mterm(INST,X,ID,I);
178                 -I, ID: mterm(INST,X,ID,I), ok(INST);
179                  Y     : ok(INST)
180                 },
181           lhspat_dom(X,Y), guess(INST).
```

Listing 3.23: rule definition for **lhspat**

3.4 Model semantics

Recall Definition 2.21. Let G be a GRAPPA-instance.

A three-valued interpretation v is a model of G iff v is total and $v = \Gamma_G(v)$.

With the rules from Section 3.3 an encoding to retrieve all models of a GRAPPA-instance can easily be defined – as shown in Listing 3.24. First all possible model-candidates are guessed – done in Line 3. Each node is either accepted or not accepted and therefore assigned to the correspondent state **in** resp. **out**.

An interpretation is not a valid model if for a state, which is not accepted, the **accept** predicate can be derived or dually if for an accepted state the **accept** predicate is not derived. This condition is expressed in the constraints in the Lines 5 and 6. So all invalid interpretations are filtered out and every found answer-set is a model of the GRAPPA-instance.

The rule in Line 8 only avoids some warnings of the ASP-solver.

As already mentioned the basic rules in Section 3.3 are designed such that they can be applied on more than one specific guess – refer to Section 2.5.7. For the model semantics the guess is executed with the id “0”. To “trigger the execution” of the rules this id must be stated by the atom **guess(0)**.

```
1 guess(0).
2
3 in(0,S) | out(0,S) :- s(S,_).
4
5 :- accept(0,X), out(0,X).
6 :- not accept(0,X),in(0,X).
7
8 ok(0) :- #false.
```

Listing 3.24: rules for the model semantics

3.5 Admissible semantics

Recall Definition 2.21. Let G be a GRAPPA-instance.

A three-valued interpretation v is admissible in G iff $v \subseteq \Gamma_G(v)$,

where $\Gamma_G(v)$ is the characteristic operator from Definition 2.26.

To compute the admissible interpretations, again all possible interpretations need to be considered. But in this case a three-valued interpretation is necessary. This guess is performed in Listing 3.25 and the three-valued interpretation is identified with index “`0`”, i. e. the atoms corresponding to this guess are **in(0,_)**, **out(0,_)**, **undec(0,_)**.

```
5 in(0,S) | out(0,S) | undec(0,S) :- s(S,_).
```

Listing 3.25: three-valued guess

To determine if a particular three-valued interpretation is indeed admissible its completions need to be considered. The completion is performed by guessing a two-valued interpretation – Line 32 of Listing 3.26 – and then assuring that the nodes of the GRAPPA-instance which are assigned to **in**, **out** by the three-valued interpretation are also assigned to **in**, **out** by the two-valued interpretation. The atoms corresponding to the two-valued guess are identified with the index "`1`" i. e. **in(1,_)**, **out(1,_)**.

The predicate **guess(0)** is missing because this guess should not be handled by the rules of the basic definitions from Section 3.3. In contrary, the predicate **guess(1)** "triggers" the execution of the rules from Section 3.3 for this guess.

The reason why in Listing 3.26 the rules use the parameter `INST`, instead of just writing `1` is, that later – for the complete semantics – these rules are also used for further guesses.

```
30 guess(1).
31
32  in(INST,S) | out(INST,S) :- s(S,_), guess(INST).
33  in(INST,S) :- in(0,S), guess(INST).
34 out(INST,S) :- out(0,S), guess(INST).
```

Listing 3.26: completion of the three-valued guess

A particular three-valued interpretation v is admissible if for every completion $v' \in [v]_c$ holds that for every node $X \in v'$ – i. e. **in(1,X)** – the atom **accept(1,X)** is derived and for every node $X \notin v'$ – i. e. **out(1,X)** – the atom **notaccept(1,X)** is derived. At this point saturation comes into play, because this condition must be verified for every completion. This check is performed for each completion v' on its own and all completions which pass the check, i. e. the atom **ok** is derived, are merged into a single interpretation by the saturation methodology.

This check is defined in the program – in Listing 3.27 – by counting all **in(1,_)** atoms which are accepted – in Line 19 – together with all **out(1,_)** atoms, which are not accepted – in Line 20. This sum must match the total number of defined nodes, i. e. all **in(0,_)**/ **out(0,_)** – but not **undec** – nodes which is retrieved in Line 16.

If there is a completion which does not satisfy the condition, it causes all other valid completions to "disappear" by the properties of the saturation technique (see Section 2.5.9). Moreover, the invalid completion is removed from the answer-sets by the rule in Line 23. The rule for **ok(1)** does not require a reformulation, because it fires also in case of saturation.

```
16 cntSelected(M) :- M = #count{X: in(0,X);Y:out(0,Y)}.
17
18 ok(1) :- #sum{
19       1, in(X):in(0,X), accept(1,X);
20       1, out(X):out(0,X), notaccept(1,X)
21       } = M, cntSelected(M).
22
23 :- not ok(1).
```

Listing 3.27: filter admissible interpretations

Finally, the saturation is performed by the rules in listing Listing 3.28. The rules ensure that every atom – affected by the second (two-valued) guess – is derived. All the rules in Listing 3.28 which derive **ok** look out of place but the necessity of them is explained in Section 2.5.9. Without them some valid answer-sets would vanish from the result.

```
41 in(INST,X) :- s(X, _), ok(INST).
42 out(INST,X) :- s(X, _), ok(INST).
43 ok(INST) :- in(INST,X), out(INST,X).
44
45 activeedge(INST,Source,Target,Label) :- e(Source,Target,Label), ok(INST).
46 ok(INST) :- activeedge(INST,Source, Target, _), out(INST,Source).
47
48 alcount(INST,NODE,L,X):- alcount_dom(NODE,L,X), ok(INST).
49 ok(INST) :- alcount(INST,NODE,L,X), alcount(INST,NODE,L,Y), X!=Y.
50
51 maxalabel(INST,NODE,NUMBER) :- maxalabel_dom(NODE,NUMBER), ok(INST).
52 ok(INST) :- maxalabel(INST,NODE,X), maxalabel(INST,NODE,Y), X!=Y.
53
54 minalabel(INST,NODE,NUMBER) :- minalabel_dom(NODE,NUMBER), ok(INST).
55 ok(INST) :- minalabel(INST,NODE,X), minalabel(INST,NODE,Y), X!=Y.
56
57 sumalabel(INST,NODE,NUMBER) :- sumalabel_dom(NODE, NUMBER), ok(INST).
58 ok(INST) :- sumalabel(INST,NODE, X), sumalabel(INST,NODE, Y), X!=Y.
59
60 distactivelabel(INST,N,X):- distactivelabel_dom(N,X), ok(INST).
61 ok(INST) :- distactivelabel(INST,NODE,X), distactivelabel(INST,NODE,Y), X!=Y.
62
63 mterm(INST,X,ID,Y..Z):- mterm_max(X,ID,Y,Z), ok(INST).
64 ok(INST) :- mterm(INST,X,ID,Y), mterm(INST,X,ID,Z), Y!=Z.
65
66 lhspat(INST,X,Y):- lhspat_dom(X,Y), ok(INST).
67 ok(INST) :- lhspat(INST,X, Y), lhspat(INST,X,Z), Y != Z.
68
69 true(INST,X) :- pattern(X), ok(INST).
70 false(INST,X) :- pattern(X), ok(INST).
71 ok(INST) :- true(INST,X), false(INST,X).
72
73 ismodel(INST,X) :- pattern(X), ok(INST).
74 nomodel(INST,X) :- pattern(X), ok(INST).
75 ok(INST) :- ismodel(INST,X), nomodel(INST,X).
76
77 accept(INST,X) :- s(X, _), ok(INST).
78 notaccept(INST,X) :- s(X, _), ok(INST).
79 ok(INST) :- accept(INST,X), notaccept(INST,X).
```

Listing 3.28: saturation rules

3.6 Complete semantics

Recall Definition 2.21. Let G be a GRAPPA-instance.

A three-valued interpretation v is complete in G iff $v = \Gamma_G(v)$.

A direct conclusion from Definition 2.21 and the definition of operator Γ_G – from Definition 2.26 – is Proposition 3.1.

Proposition 3.1. An admissible interpretation v is complete, i. e. $v = \Gamma_G(v)$, if for every undecided variable a in v there exist completions $v'_1, v'_2 \in [v]_c$ s. t.

- $\alpha(a)(v'_1) = t$ and
- $\alpha(a)(v'_2) = f$.

The encoding presented in this section is working on top of the encoding of the admissible semantics as described in Section 3.5, because only admissible interpretations are candidates for a complete interpretation.

The characterization of complete interpretations in Proposition 3.1 is used to remove interpretations, which are not complete, from the admissible interpretations. This characterization implies that – in the worst case – two interpretations must be guessed for every undecided node in a given GRAPPA-instance. The rule in Line 3 counts the number of undecided nodes and the rule in Line 4 of Listing 3.29 ensures that enough guesses are performed.

In Line 6 the check from Proposition 3.1 is executed for every undecided node, i. e. two different guesses `G1, G2: G1 != G2` are searched, which are part of the guesses done in Line 4 – i. e. `G1>=2, G2>=2` – and where the node is accepted in one guess and not accepted in the other guess. Finally, the interpretation is complete – as stated in Line 8 – if the number of such nodes – marked with **undef**– equals the number of undecided nodes.

The id of the guesses starts at 2, because **guess(1)** is reserved for the admissible encoding.

```
3 numUndec(M) :- M = #count{X:undec(0,X)}.
4 guess (2..X) :- X=2*M+1, numUndec(M).
5
6 undef(X) :- accept(G1,X), notaccept(G2,X), undec(0,X), G1 != G2, G1 >= 2,
    ↪ G2>=2.
7
8 complete :- M = #sum{
9            1,X: undef(X)
10          }, numUndec(M).
11
12
13 :- not complete.
```

Listing 3.29: rules for complete sematics

CHAPTER 4

Dynamic encodings

In this section the *dynamic* encoding is described to evaluate GRAPPA-instances under admissible, complete or preferred semantics. The name “dynamic” was chosen because the encoding is generated for each GRAPPA-instance separately. This generation must be executed by a separate tool which takes a GRAPPA-instance as input and produces the ASP encoding. In this case this tool is implemented as part of GrappaVis which is presented in Chapter 6.

The main advantage of this approach is, as discussed in Section 2.6.4, that Π_3^{P} complete problems can be encoded. Hence it is also possible to provide an efficient encoding for the preferred semantics, which is not possible for the static encoding.

The big difference of this kind of encoding is that the GRAPPA-instance is not represented as a set of facts – as it is in the case of the *static encoding*. This approach encodes the instance directly into a set of rules.

In this chapter the encodings for the admissible, complete and preferred semantics are presented. The basic idea of the encoding – which is similar for all semantics – is discussed within the section of the admissible semantics, as well as the rules to handle the acceptance patterns of the GRAPPA-instances. The encoding for the complete and preferred semantics then extend the encoding for the admissible semantics to retrieve the corresponding interpretations.

4.1 Admissible semantics

4.1.1 Overview

A direct conclusion from Definition 2.21 and the definition of operator Γ_G – from Definition 2.26 – is Proposition 4.1.

Proposition 4.1. An interpretation v is admissible in G if for every $s \in S$ it holds that

- if $s \in v$ then there is no $v' \in [v]_c$ s. t. $\alpha(s)(m_s^{v'}) = f$
- if $\neg s \in v$ then there is no $v' \in [v]_c$ s. t. $\alpha(s)(m_s^{v'}) = t$

The basic idea of this encoding is to define two rules for every node $s \in S$, which derive for a given interpretation v

- an atom **sat(s)** if there is a completion $v' \in [v]_c$ s. t. the acceptance condition does hold.
- an atom **unsat(s)** if there is a completion $v' \in [v]_c$ s. t. the acceptance condition does not hold.

The trick is that rules for **sat(s)** resp. **unsat(s)** are designed to cover the idea from Section 2.5.8, i. e. every rule builds all completions within its own rule body and checks if any $v' \in [v]_c$ satisfies resp. not satisfies the acceptance condition for the node s.

With the rules for **sat** resp. **unsat** the check of Proposition 4.1 can easily be done, because it boils down to the addition of constraints to the program which prohibit for an interpretation v the existence of

- an atom **sat(s)** if $\neg s \in v$.
- an atom **unsat(s)** if $s \in v$.

4.1.2 Encoding

For the encodings of the admissible semantics a set of facts – comprising **arg** and **leq** predicates – is required.

arg(`node`)

`node` A node of the instance.

description: This predicate is used to specify the set of nodes of the GRAPPA-instance under evaluation.

leq(`l`, `h`)

description: This predicate defines the information order of the truth values $\{0, 1, u\}$ as defined in Section 2.1.2, by adding the four facts **leq(u,0)**, **leq(u,1)**, **leq(0,0)**, **leq(1,1)**. The predicate is used to compute completions of a given interpretation.

To retrieve all admissible interpretations all possible three-valued interpretations have to be guessed, which is done as shown in Listing 4.1. The rules ensure that every node is assigned to one of the three values `{u, 1, 0}` – denoting “undecided”, “true” and “false” – which is stored in the atom **ass**.

```
1 ass(X,0) :- not ass(X,1), not ass(X,u), arg(X).
2 ass(X,1) :- not ass(X,0), not ass(X,u), arg(X).
3 ass(X,u) :- not ass(X,0), not ass(X,1), arg(X).
```

Listing 4.1: guessing interpretations

Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance and $S = \{s_1, \dots, s_k\}$ the set of all arguments of G. As already mentioned in Section 4.1.1, rules are required to derive $\mathbf{sat(}s_i\mathbf{)}$ resp. $\mathbf{unsat(}s_i\mathbf{)}$ for each node s_i of the GRAPPA-instance. These rules consist of two parts. First there is the "guessing"-part presented in Eqs. (4.1) and (4.2).

$$ass(s_1, X1), \dots, ass(s_k, Xk), \tag{4.1}$$
$$leq(X1, S1), \dots, leq(Xk, Sk) \tag{4.2}$$

In the Eq. (4.1) the three-valued interpretation, computed in Listing 4.1, is stored into the local variables Xi. In the Eq. (4.2) the **leq** predicates generate the completions. The definition of the predicate **leq** ensures that

- if $X_i = 1$ or $X_i = 0$, then $S_i = X_i$.
- if $X_i = u$ then $S_i \in \{0, 1\}$.

This coerces the rule to evaluate all possible completions of the three-valued interpretation given by **ass**.

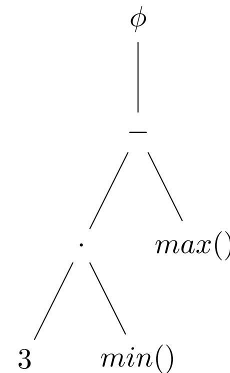
The second part of the rule evaluates the acceptance pattern of the node s for which the rule is written. For this task the acceptance pattern is split into its basic components, i. e. the functions of the GRAPPA pattern-language (`#`, `min`, `max`, ...), arithmetic functions and boolean functions, which are building the syntax-tree of the pattern. Every basic component is converted into an ASP atom which stores the result in a new local variable to make the result available for the next computation step. This sounds more complicated than it actually is as demonstrated in the small Example 4.1.

Example 4.1. Let $\phi := 3 \cdot min() - max()$ an acceptance pattern.

The syntax-tree is shown at the right side. The atoms for the acceptance pattern ϕ are generated bottom up.

The atoms generated for ϕ are:

$$\begin{aligned} Var1 &= \#min\,\{\cdots\}\,, \\ Var2 &= 3 * Var1, \\ Var3 &= \#max\,\{\cdots\} \\ Var4 &= Var2 - Var3 \end{aligned}$$



The atoms for the evaluation of *min*/ *max* are only snippets. The exact definition is given later in this chapter. △

Let (S, E, L, λ, π) be a GRAPPA-instance, n the index of the node $s_n \in S$ whose acceptance condition should be evaluated and $P(n) = \{i \mid (s_i, s_n) \in E\}$ the set of all indices of predecessors of s_n, i. e. $\{s_i \mid i \in P(n)\}$ is the set of nodes s_n depends on.

The rules to convert the functions of the GRAPPA pattern-language are given in Eq. (4.3) to Eq. (4.12). The variable Var_k is instantiated for each application of a rule, i. e. the index k is unique for each variable.

Some remarks on the notation for the conversion rules:

- The conversion is denoted as <fct> → <atom>, where <fct> represents the function to convert and <atom> the atom which is generated for <fct>.
- The indices a and z, are representing the first and the last element of the set $P(n)$. An expression like Sa; ... ;Sz indicates that all elements of $P(n)$ are iterated. For example let $P(n) = \{3, 6, 9, 12\}$. The expression $\{Sa$; ... ;$Sz\}$ is decoded as $\{S3; S6; S9; S12\}$.
- l_a represents the label of the edge $(s_a, s_n) \in E$
- *minDom*, *maxDom* represents the minimum and maximum of the domain for the given GRAPPA-instance. These values are added to the set which the ASP-aggregates `#min` resp. `#max` evaluate, to avoid that the aggregates yields ∞ resp. $-\infty$ in case that $P(n)$ is empty.

$$\#(l) \rightarrow Var_k = \#\text{sum} \quad \{Sa, s_a;\ \ldots\ ;Sz, s_z\} \tag{4.3}$$
$$\#_t(l) \rightarrow Var_k = \#\text{count} \quad \{s_a;\ \ldots\ ;s_z\} \tag{4.4}$$
$$\min() \rightarrow Var_k = \#\text{min} \quad \{maxDom; l_a : Sa = 1;\ \ldots\ ;l_z : Sz = 1\} \tag{4.5}$$
$$\min_t() \rightarrow Var_k = \#\text{min} \quad \{maxDom; l_a;\ \ldots\ ;l_z\} \tag{4.6}$$
$$\max() \rightarrow Var_k = \#\text{max} \quad \{minDom; l_a : Sa = 1;\ \ldots\ ;lz : Sz = 1\} \tag{4.7}$$
$$\max_t() \rightarrow Var_k = \#\text{max} \quad \{minDom; l_a;\ \ldots\ ;l_z\} \tag{4.8}$$
$$\text{sum}() \rightarrow Var_k = \#\text{sum} \quad \{l_a, s_a : Sa = 1;\ \ldots\ ;l_z, s_z : Sz = 1\} \tag{4.9}$$
$$\text{sum}_t() \rightarrow Var_k = \#\text{sum} \quad \{l_a, s_a;\ \ldots\ ;l_z, s_z\} \tag{4.10}$$
$$\text{count}() \rightarrow Var_k = \#\text{count} \quad \{l_a : Sa = 1;\ \ldots\ ;l_z : Sz = 1\} \tag{4.11}$$
$$\text{count}_t() \rightarrow Var_k = \#\text{count} \quad \{l_a;\ \ldots\ ;l_z\} \tag{4.12}$$

The variables Si are either 0 or 1, and $Si = 1$ is indicating that the edge (s_i, s_n) is an active edge. Therefore, Si can be used to count the number of active edges in Eq. (4.3). To make the set-entry unique for each node Si is coupled to the parent node as a tuple (Si, s_i). For the other rules an explicit condition for an active edge is necessary. This is

achieved by adding the constraint $Si = 1$ to the corresponding set element in Eqs. (4.5), (4.7), (4.9) and (4.11).

The functional combinations are evaluated by multiplying the coefficients of the terms to the function results in Eq. (4.13) and then summing these results up – step by step – in Eq. (4.14).

$$\text{const} \cdot Var_j \quad \rightarrow \quad Var_k = \text{const} * Var_j \tag{4.13}$$
$$Var_i + Var_j \quad \rightarrow \quad Var_k = Var_i + Var_j \tag{4.14}$$

The next step is to evaluate the comparison of the result of the functional combination to the constant. Here a trick is applied, because an atom of the form $Var_k = (Var_i < Var_j)$ is not allowed. So a sum-aggregate is used to calculate the truth value in Eq. (4.15).

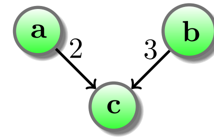
$$Var_i \circ \text{const} \quad \rightarrow \quad Var_k = \#\text{sum}\{1 : Var_i \circ \text{const}\} \quad \circ \in \{<, <=, =, !=, >=, >\} \tag{4.15}$$
$$Var_i \text{ and } Var_j \quad \rightarrow \quad Var_k = Var_i \,\&\, Var_j \tag{4.16}$$
$$Var_i \text{ or } Var_j \quad \rightarrow \quad Var_k = Var_i \,?\, Var_j \tag{4.17}$$
$$Var_i \text{ xor } Var_j \quad \rightarrow \quad Var_k = Var_i \,\hat{}\, Var_j \tag{4.18}$$
$$sat \quad \rightarrow \quad Var_i = 1 \tag{4.19}$$
$$unsat \quad \rightarrow \quad Var_i = 0 \tag{4.20}$$

To propagate the truth-values through the arbitrary boolean functions it becomes handy that at least some ASP-solvers have built-in bitwise boolean operations. Because only 0 and 1 are used as truth-values these build-in operations suffice to calculate the corresponding boolean functions in Eqs. (4.16) to (4.18).

The last step depends on whether the **sat** or **unsat** predicate should be derived. In the first case the rule Eq. (4.19) in the latter case the rule Eq. (4.20) is added. In this case Var_i is no new variable but the variable which holds the result of the acceptance pattern.

Example 4.2.

Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance, with
$S = \{a, b, c\}$, $E = \{(a, c), (b, c)\}$,
$\lambda = \{(a, c) \mapsto 2, (b, c) \mapsto 3\}$ and
$\pi = \{s \mapsto f \mid s \in S\}$,



where

$$f := 4 * \#(2) - 2 * \text{sum}() = 14 \text{ xor count}() \geq 0 \text{ or min}() \neq 1 \tag{4.21}$$
$$= (4 * \#(2) + (-2) * \text{sum}() = 14 \text{ xor } (\text{count}() \geq 0 \text{ or min}() \neq 1)) \tag{4.22}$$

Equation (4.22) illustrates the execution order of the operations.

```
1 sat(c) :- ass( a,X1), ass( b,X2), ass( c,X3),
2           leq(X1,S1), leq(X2,S2), leq(X3,S3),
3           Var1 = #sum{S1,a},                % #(2)
4           Var2 = 4 * Var1,                  % 4*#(2)
5           Var3 = #sum{2,a; 3,b},            % sum()
6           Var4 = -2 * Var3,                 % -2 * sum()
7           Var5 = Var2 + Var4,               % 4*#(2) - 2 * sum()
8           Var6 = #sum{1:Var5=14},           % 4*#(2) - 2 * sum() == 14
9           Var8 = #count{2:S1=1;3:S2=1},     % count()
10          Var9 = #sum{1:Var8>=0},           % count() >= 0
11         Var10 = #min{2:S1=1;3:S2=1\},      % min()
12         Var11 = #sum{1:Var8!=0},           % min() != 1
13         Var12 = Var9 ? Var11,              % count() >= 0 or min() != 1
14         Var13 = Var6 ^ Var12,              % 4*#(2) - 2 * sum() == 14 xor
15                                            %     count() >= 0 or min() != 1
16         Var13 = 1.
17
18 unsat(c) :- ass( a,X1), ass( b,X2), ass( c,X3),
19          leq(X1,S1), leq(X2,S2), leq(X3,S3),
20          Var1 = #sum{S1,a},                % #(2)
21          Var2 = 4 * Var1,                  % 4*#(2)
22          Var3 = #sum{2,a; 3,b},            % sum()
23          Var4 = -2 * Var3,                 % -2 * sum()
24          Var5 = Var2 + Var4,               % 4*#(2) - 2 * sum()
25          Var6 = #sum{1:Var5=14},           % 4*#(2) - 2 * sum() == 14
26          Var8 = #count{2:S1=1;3:S2=1},     % count()
27          Var9 = #sum{1:Var8>=0},           % count() >= 0
28         Var10 = #min{2:S1=1;3:S2=1\},      % min()
29         Var11 = #sum{1:Var8!=0},           % min() != 1
30         Var12 = Var9 ? Var11,              % count() >= 0 or min() != 1
31         Var13 = Var6 ^ Var12,              % 4*#(2) - 2 * sum() == 14 xor
32                                            %     count() >= 0 or min() != 1
33         Var13 = 0.
```

Listing 4.2: rule for **sat(c)** and **unsat(c)**

The rules for **sat(c)** and **unsat(c)** are exactly the same, with only the last line – Line 16 resp. Line 33 – being different. △

Finally, the rules in Listing 4.3 remove all interpretations which are not admissible.

```
1 :- arg(S), ass(S,1), unsat(S).
2 :- arg(S), ass(S,0), sat(S).
```

Listing 4.3: remove invalid interpretations

4.2 Complete semantics

The encoding is based on the same consideration as for the static counterpart in Section 3.6, especially the Proposition 3.1.

Because every complete semantics is admissible the following rules are used in addition to the rules described in Section 4.1. The rules to check the existence of the two completions v_1', v_2' are basically the same as the rules for **sat**/ **unsat** in Section 4.1, but in this case these two rules are merged into one, i. e. two independent completions are generated within one rule. So in the end there is an additional rule for each statement s_i which derives an atom **undec(**s_i**)** if v_1, v_2 can be found. So the first part of the rule looks like the Eqs. (4.23) to (4.26).

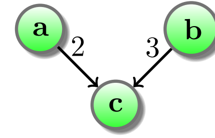
$$ass(s_i, u), \tag{4.23}$$
$$ass(s_1, X1), \ldots, ass(s_k, Xk), \tag{4.24}$$
$$leq(X1, S1), \ldots, leq(Xk, Sk) \tag{4.25}$$
$$leq(X1, T1), \ldots, leq(Xk, Tk) \tag{4.26}$$

Noteworthy is that Eq. (4.24) is exactly the same as Eq. (4.1) as well as Eq. (4.25) and Eq. (4.2) are equal. Equation (4.23) ensures the rule fires only if the statement s is undecided.

In Eq. (4.25) all possible interpretations for the completion v_1' are generated. Equation (4.26) does the same but stores the result it into the variables Ti. The completions have to be evaluated and this is done in exactly the same way as in Section 4.1 by Eq. (4.3) to Eq. (4.18). The only extension is that the occurrences of Si must be replaced by Ti when applying the rules for the second interpretation. The rules for the first interpretation are closed by Eq. (4.19), i. e. the rules check whether v_1' satisfies the acceptance pattern of the node, and the rules for the second interpretation v_2' are closed by Eq. (4.20), i. e. the rule checks whether v_2' invalidates the acceptance pattern of the node.

Example 4.3.

Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA-instance, with
$S = \{a, b, c\}$, $E = \{(a, c), (b, c)\}$,
$\lambda = \{(a, c) \mapsto 2, (b, c) \mapsto 3\}$ and
$\pi = \{s \mapsto f \mid s \in S\}$,
where



$$f := 2 * \text{count}() \neq 5 \tag{4.27}$$

```
1 undec(c) :- ass(c,u),
2            ass( a,X1), ass( b,X2), ass( c,X3),
3            leq(X1,S1), leq(X2,S2), leq(X3,S3),
4            leq(X1,T1), leq(X2,T2), leq(X3,T3),
5            Var1 = #count{2:S1=1;3:S2=1},     % count()
6            Var2 = 2 * Var1,                   % 2 * count()
7            Var3 = #sum{1:Var2 != 5},          % 2 * count() != 5
8            Var3 = 1,
```

```
9          Var11 = #count{2:T1=1;3:T2=1},     % count()
10         Var12 = 2 * Var11,                 % 2 * count()
11         Var13 = #sum{1:Var12 != 5},        % 2 * count() != 5
12         Var13 = 0.
```

Listing 4.4: rule for **undec(c)**

△

Finally, the rule in Listing 4.5 removes all interpretations which have an undecided variable but **undec** was not derived. This yields all complete interpretations.

```
1 :- not undec(X), ass(X,u).
```

Listing 4.5: remove invalid interpretations

4.3 Preferred semantics

Recall Definition 2.21:

A three-valued interpretation v is preferred in G iff v is subset-maximal admissible in G.

To check the subset-maximality saturation is necessary again. But the application for the dynamic encoding is not as difficult as in the static encoding. The problem of the static encoding was that the aggregates of the rules where depending on an atom, namely **activeedge**. As a result, the naive formulation of a rule fires if not saturated, but does not fire if saturated, what causes problems as described in Section 2.5.9. In the dynamic encoding the aggregates only depend on variables which are within the rule of the aggregate itself, i. e. the aggregate is not affected by saturation in a way, that the rule-body becomes invalid after saturation.

For that reason, the rules from the admissible encoding can stay unaltered. The admissible encoding yields all admissible interpretations $\mathcal{I}$. Now every interpretation $v \in \mathcal{I}$ has to be checked if it is subset-maximal. This is done by guessing all completions $[v]_c$ of v and checking whether there is any $v' \in [v]_c$ which is admissible. If a particular $v' \in [v]_c$ is admissible, saturation is performed, otherwise not. Then only those $v \in \mathcal{I}$ have to be removed which are not saturated, i. e. are not subset-maximal.

To this end all possible completions are generated by the rules in Listing 4.6. Every atom, which is introduced for checking the completions, is labeled with a prefix “2” to avoid clashes with the atoms used to compute the admissible interpretations. So the predicate **ass2** stores the completion. In Line 3 all possible interpretations are generated. The rules in the Lines 1 and 2 assure that generated interpretations are indeed valid completions.

```
1  ass2(S,0):-ass(S,0)
2  ass2(S,1):-ass(S,1)
3  ass2(S,1)|ass2(S,0)|ass2(S,u):-ass(S,u).
```

Listing 4.6: rules for completion

There is one flaw in generating the completions, namely that a completion is generated which is exactly the same as the original interpretation. This interpretation would be found to be admissible again and therefore no interpretation would be preferred. Therefore, this special case is taken care of in the rule in Listing 4.7. It checks the number of undecided nodes in the original and completed interpretation. If they match, saturation is performed, so this case will not invalidate the result. Normally the result of the aggregate is exactly zero in such a case. But in case of saturation there are much more undecided nodes and therefore the result becomes negative. To keep the rule firing in case of saturation the comparator is "$\leq$".

```
1   saturate:-0<=#sum{-1,S:ass(S,u);1,S:ass2(S,u)}.
```

Listing 4.7: taking care of case if the completion is the same as the original

Then the completions have to be checked if they are admissible. This is done by the application of the same rules as described in Section 4.1. That means the rules for **sat** and **unsat** are inserted once more, just with changed predicates to "access" the new completion. So **sat** is replaced by **sat2**, **unsat** by **unsat2** and **ass** by **ass2**.

The rules for enabling saturation are given in Listing 4.8. They derive **saturate** if the completion is not admissible.

```
1   saturate :- arg(S), ass2(S,1), unsat2(S).
2   saturate :- arg(S), ass2(S,0), sat2(S).
```

Listing 4.8: rules for starting saturation

The rules for saturation the atoms into the answer-set are given in Listing 4.9. The necessity of the rules in the Lines 6 and 7 is explained in Section 2.5.9.

```
1   ass2(S,0) :- arg(S),saturate.
2   ass2(S,1) :- arg(S),saturate.
3   ass2(S,u) :- arg(S),saturate.
4   sat2(S) :- arg(S),saturate.
5   unsat2(S) :- arg(S),saturate.
6   saturate :- ass(S,0),sat2(S),unsat2(S).
7   saturate :- ass(S,1),sat2(S),unsat2(S).
```

Listing 4.9: rules for saturation

Finally, it is necessary to remove all answer-sets which are not saturated and therefore not preferred. This is done by the rule in Listing 4.10.

```
1   :- not saturate.
```

Listing 4.10: remove not saturated results

CHAPTER 5

Graphical user interface

In this chapter GrappaVis is introduced which is a graphical tool to specify and evaluate GRAPPA-instances. Basically GrappaVis is a graph-editor based on the *JGraphX* framework – available in [JGr16] – which is extended by a toolbox to evaluate GRAPPA-instances.

For the different evaluation methods GrappaVis also incorporates the necessary tools to (pre)process the instance, i. e. GrappaVis manages the export of the GRAPPA-instance into the required format for the chosen evaluation method and is also capable of converting GRAPPA to ADF-instances and the other way round.

This chapter starts with an overview of the different parts of the program and is followed by a tutorial which shows how to specify and evaluate GRAPPA-instances with GrappaVis. To define GRAPPA-instances, acceptance patterns for the nodes have to be defined. To this end the syntax is presented which is used to specify acceptance patterns for GrappaVis.

The methods for evaluating GRAPPA-instances can be configured to some extent. The property file for this configuration is explained in Section 5.7.

Finally, there is a small section which discusses the design of the program.

Notation

Many tasks in GrappaVis are performed by *drag and drop*. The start of the drag is referred to as *drag-source* and the end of the drag as *drop-position* or as *drop-target*.

5.1 Parts of the user interface

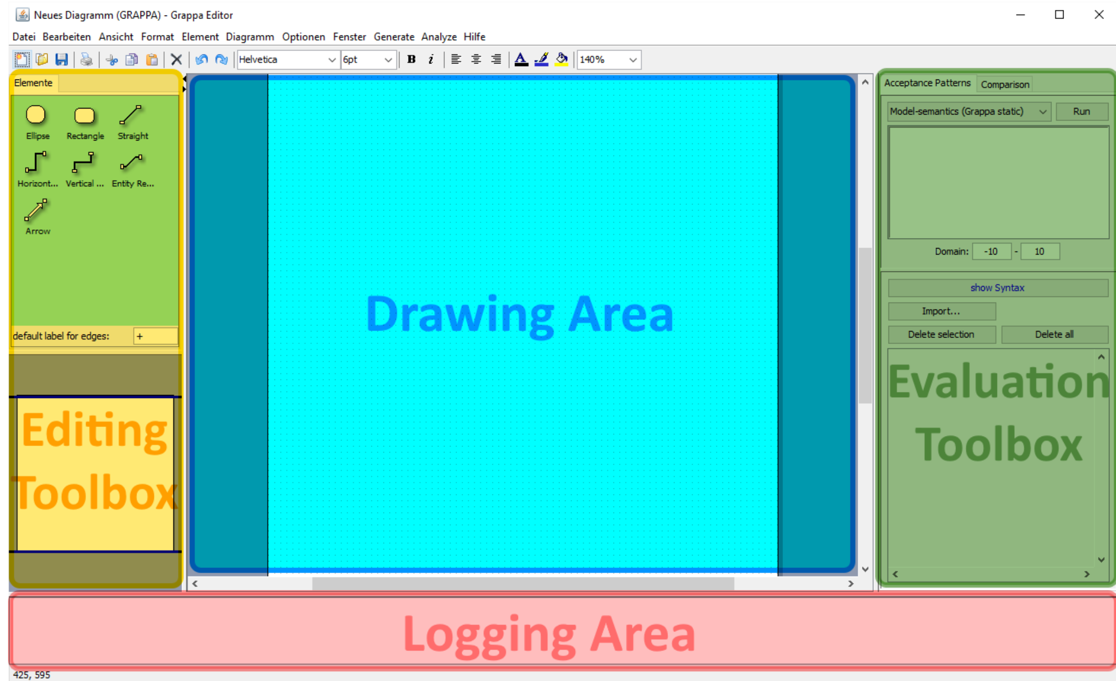


Figure 5.1: the editor

Figure 5.1 depicts a screenshot of GrappaVis with a highlighting of the different areas.

In the middle there is the *drawing area*, where the graph corresponding to the GRAPPA-instance is drawn.

At the bottom there is the *status bar* and above there is the *logging area*. In the logging area error-messages or other information are shown if problems occur during runtime. The logging area can be cleared, i. e. all entries are removed, by double-clicking with the right mouse-button on an entry of the list.

5.1.1 Editing toolbox

On the left side the toolbox for specifying and editing graphs is located. The different elements of the toolbox are highlighted in Fig. 5.2.

In the upper half of the toolbox the *template panel* is located. This panel provides different templates for nodes and edges which can be used to draw a graph.

In the lower half of the toolbox an *overview panel* of the drawing area is located. This panel shows the content of the whole drawing area and becomes useful if the graph becomes large. The blue rectangle in the overview panel depicts the current visible area of the drawing area.

In the middle, between the template and the overview panel, the default label for new edges can be defined.

5.1.2 Evaluation toolbox

The *evaluation toolbox* is on the right side of the window. This toolbox comprises two sections, namely the *acceptance pattern section*, – shown in Fig. 5.3 – and the *comparison section*– shown in Fig. 5.4.

In the upper part of the acceptance pattern section the *evaluation panel* is located. This panel is used to evaluate the GRAPPA-instance in the drawing area and is described in Section 5.4 in detail.

In the lower part of the acceptance pattern section the *acceptance pattern panel* is located. This panel is used to import acceptance patterns and to assign them to nodes in the drawing area. This panel is described in detail in Section 5.3.

The comparison section is used to execute two different evaluation methods on the GRAPPA-instance in the drawing area and to compare the results. A detailed description is given in Section 5.4.

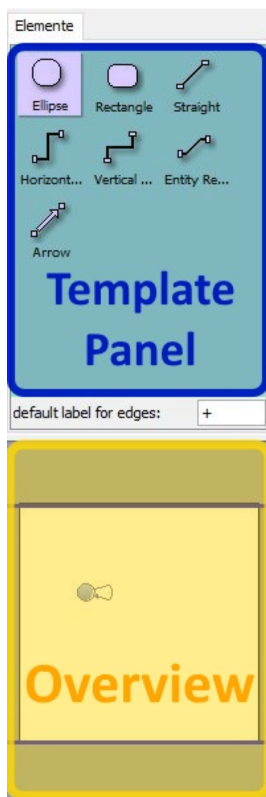


Figure 5.2: editing toolbox

Figure 5.3: acceptance pattern section

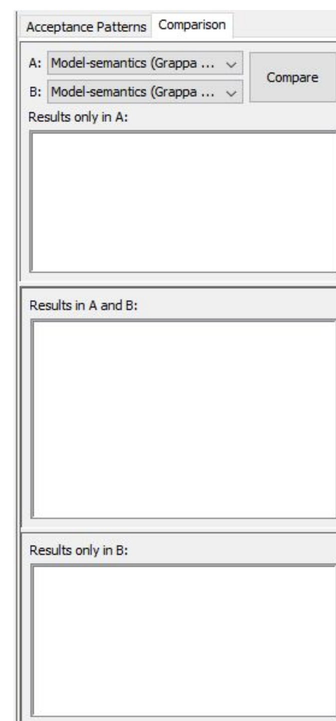


Figure 5.4: comparison section

5.2 Drawing a graph

Drawing a graph in GrappaVis is quite intuitive. To start, a node-template from the template panel on the left is dragged into the drawing area – shown in Fig. 5.5.

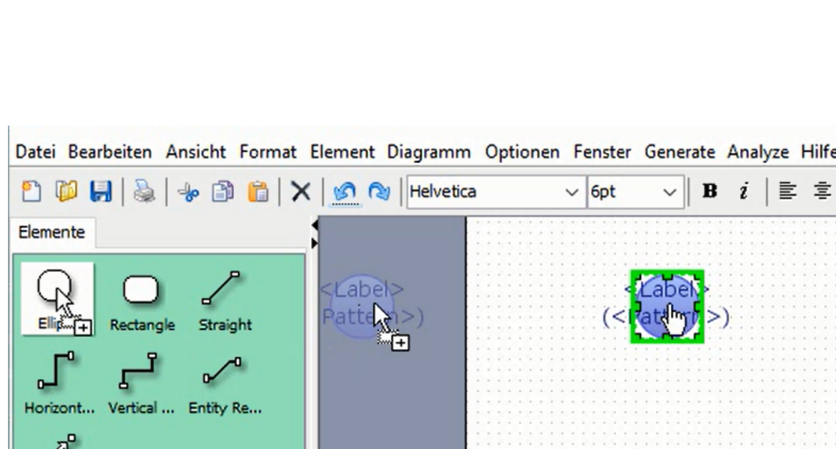


Figure 5.5: dragging a node into the drawing area

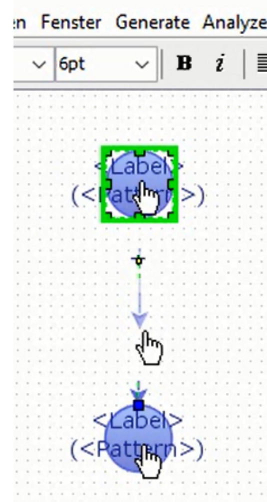


Figure 5.6: adding an edge with a new node

Drag and drop of a node, which is already in the drawing area, results in two different actions depending on the mouse cursor.

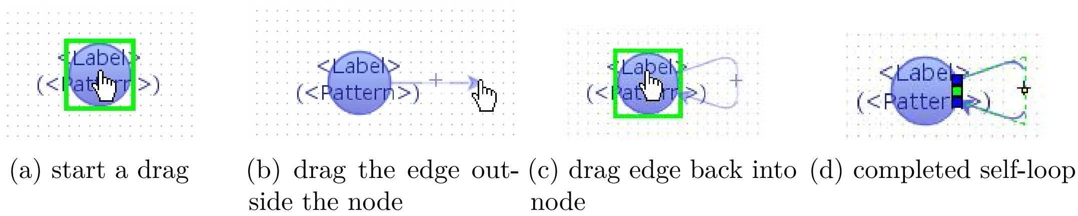
- If the cursor is approximately in the middle of the node, the cursor becomes a pointing hand and the node is surrounded by a green rectangle as shown in Fig. 5.7a. In this mode a new edge between the drag-source and the drop-target is inserted.

(a) pointing hand cursor

(b) 4-way-cross

Figure 5.7: different types of cursors

If the drop-target is a node, only the edge is inserted. If drag-source and drop-target are the same node a self loop is inserted – Fig. 5.8. If there is no node at the drop-position a new node is inserted at the drop-position and the edge is connecting the drag-source and the new node – shown in Fig. 5.6. This mode is very comfortable to quickly draw a graph.



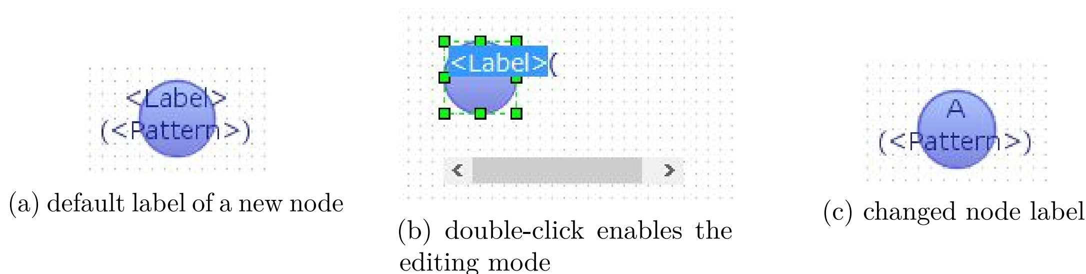
(a) start a drag (b) drag the edge outside the node (c) drag edge back into node (d) completed self-loop

Figure 5.8: drawing a self-loop

- If the cursor is approximately over the edge of the node, the cursor becomes a “4-way-cross” as shown in Fig. 5.7b. In this mode only the selected node is moved by drag and drop. If there are edges connected to the node, the starting point of these edges are moved together with the node, i. e. the edges will not become disconnected if the node is relocated.

5.2.1 Labelling nodes and edges

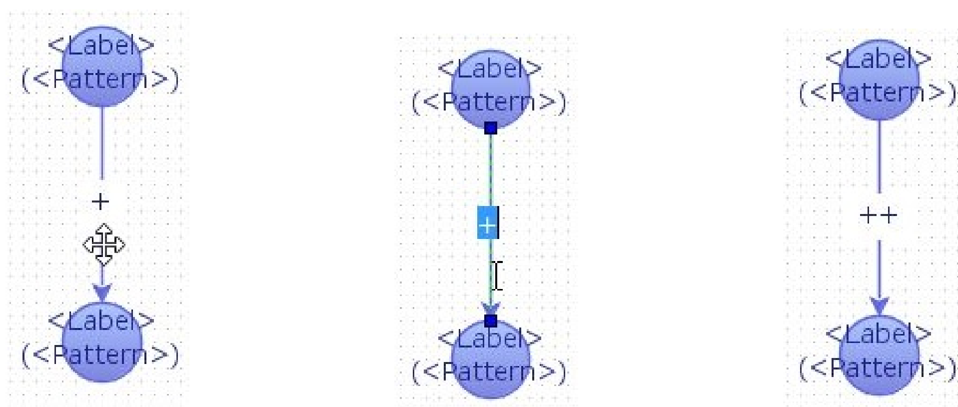
The label of a node consists of two parts: The actual label and the id of the acceptance pattern and are displayed under each other as shown in Fig. 5.9a. The id of the acceptance pattern is enclosed by brackets to distinguish it from the label of the node. By double clicking on the node the *editing mode* of the node is activated. If the editing mode is activated the node label is selected automatically such that it is possible to start typing and replacing only the label of the node but not the id of the acceptance pattern as shown in Fig. 5.9b. The id of the acceptance pattern can also be changed in editing mode as long as the id is still enclosed by brackets. But later a more convenient way to assign acceptance patterns to a node is presented. To exit the editing-mode either press the “Enter”-key or click with the mouse into an empty area of the drawing area.



(a) default label of a new node (b) double-click enables the editing mode (c) changed node label

Figure 5.9: labelling of a node

Changing an edgelabel works similar to a node. To enter the *editing mode* execute a double-click on the edge – Fig. 5.10a. To exit the editing mode either press the “Enter”-key or click anywhere on the drawing area.



(a) double-click on edge to enable editing mode

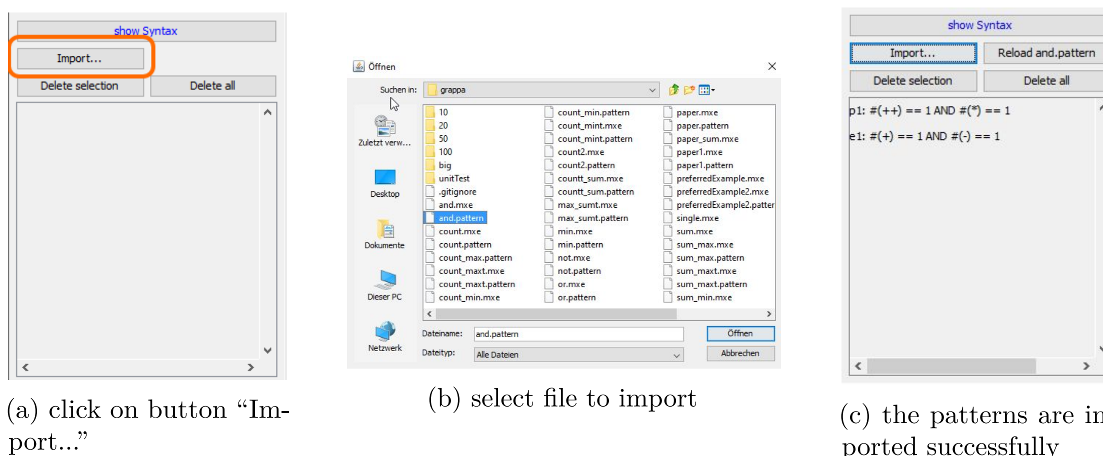
(b) editing mode activated

(c) changed edgelabel

Figure 5.10: changing an edgelabel

5.3 Handling acceptance patterns

To assign acceptance patterns to the nodes of the graph the patterns have to be imported. This is easily accomplished by pressing the “Import...”-button in the acceptance pattern panel –Fig. 5.11a. After selecting the file containing the patterns – Fig. 5.11b - the file is parsed and checked for syntax-errors. If no errors occurred the patterns are shown in the list below the buttons and a new button “Reload <filename>” becomes visible – Fig. 5.11c. As the name suggests this button reloads the file, which was imported previously.



(a) click on button “Import...”

(b) select file to import

(c) the patterns are imported successfully

Figure 5.11: importing acceptance patterns

In the list only acceptance patterns are shown but no definitions of variables. Therefore, if acceptance patterns use a variable, the definitions of the variables are shown in a small

window, which appears if the mouse-cursor is located over an acceptance pattern – as shown in Fig. 5.12.

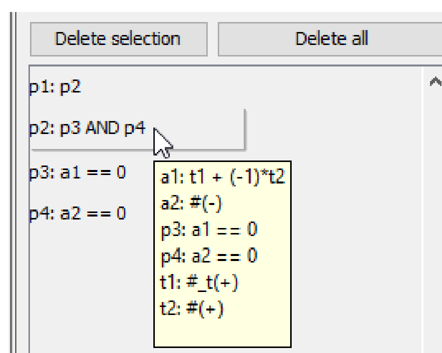
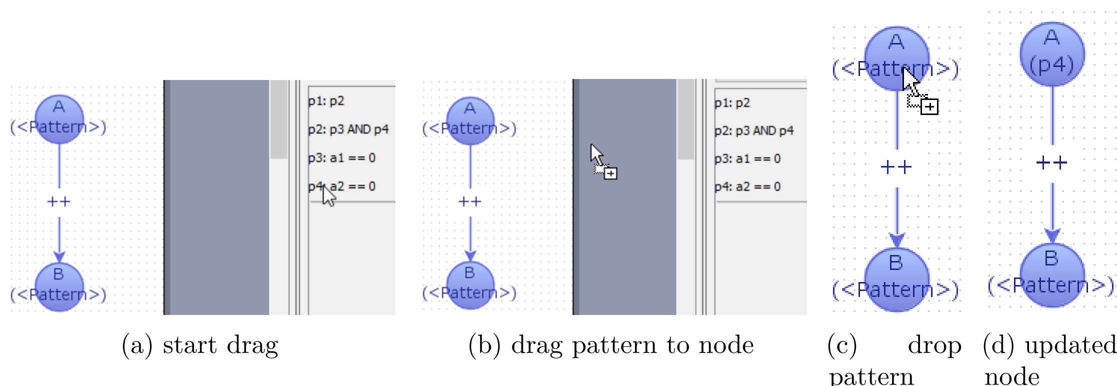


Figure 5.12: a tooltip shows the definition of the used variables

To assign a pattern to the node just drag the pattern from the list and drop it onto the node as shown in Fig. 5.13.



(a) start drag (b) drag pattern to node (c) drop pattern (d) updated node

Figure 5.13: assigning an acceptance pattern to a node

5.4 Evaluation of GRAPPA-instances

If a GRAPPA-instance is in the workspace it can be evaluated. First an evaluation method has to be chosen from the drop-down list at the top of the evaluation panel as shown in Fig. 5.14.

Additionally, the domain of the GRAPPA-instance under evaluation must be adapted. This is done by the fields given in Fig. 5.15. The values are used in the static encoding for the atoms **minDom** resp. **maxDom**. as described in Section 3.1, as well as in the dynamic encoding as described in Section 4.1.2 for the values *minDom*, *maxDom*.

To start the evaluation the button “Run” has to be pressed as shown in Fig. 5.16a.

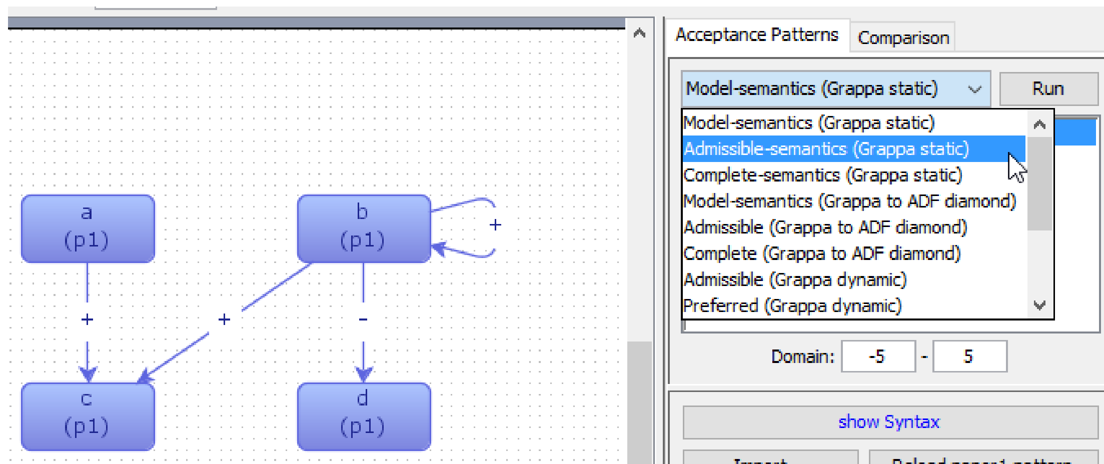


Figure 5.14: select an evaluation method

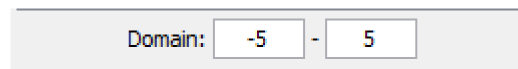
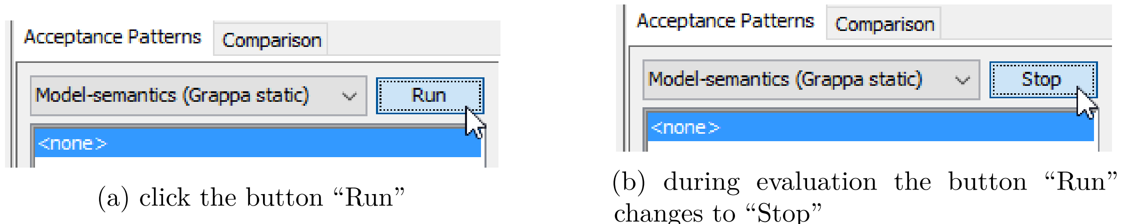


Figure 5.15: adjust the domain for the GRAPPA-instance

The button “Run” is replaced by a “Stop”-button during evaluation – shown in Fig. 5.16b. When the evaluation is finished the button “Stop” is again replaced by the original “Run”-button. For small GRAPPA-instances the change is hardly perceivable, because the evaluation takes less than a second. But for large GRAPPA-instances evaluations can take much longer. In case of time-consuming evaluations, the button “Stop” can be used to interrupt the evaluation.



(a) click the button “Run”

(b) during evaluation the button “Run” changes to “Stop”

Figure 5.16: starting the evaluation

If the evaluation is successful, the results are displayed in the list in the evaluation panel as shown in Fig. 5.17. To distinguish between accepted and not accepted nodes, not accepted nodes are preceded by a “-”. Nodes which are undefined are not listed in the result. If there is an empty result, i. e. all nodes are undefined, the entry “<empty>” is added to the list of the results. The entry “<none>” is a default entry of the list. It is used to deselect any other result and to repaint the graph in its default-colors.

By selecting a result in the list the graph in the drawing area is repainted according to

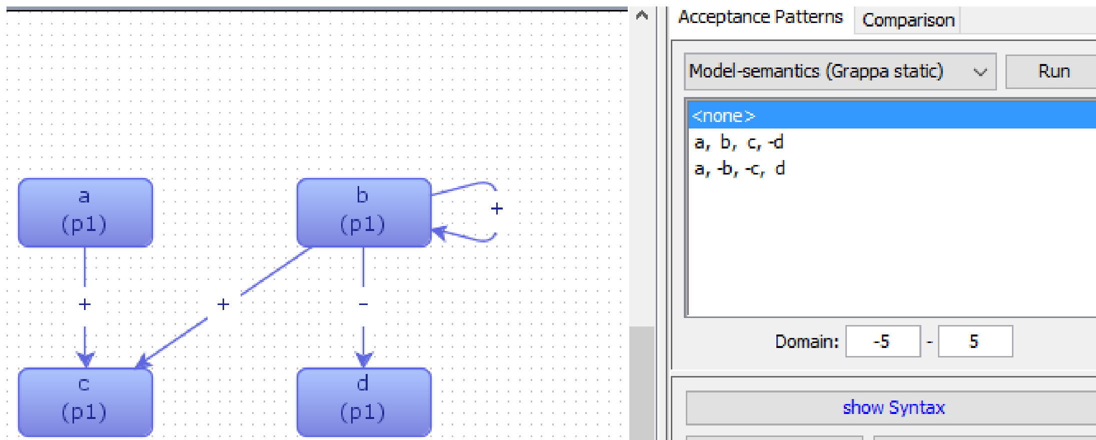


Figure 5.17: evaluation finished – button “Stop” changed back to “Run”

the selected result as shown in Fig. 5.18. Accepted nodes become green, not accepted nodes become red and undefined nodes are shown in the default color – blue.

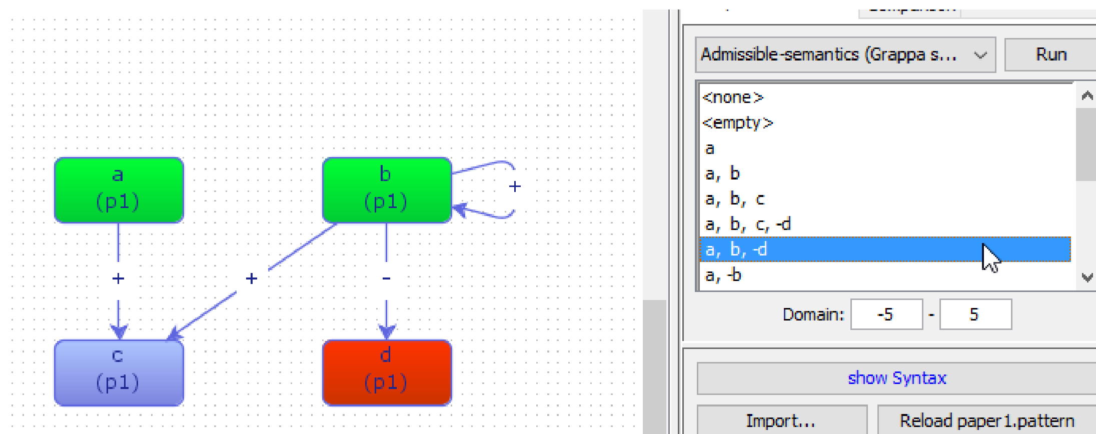


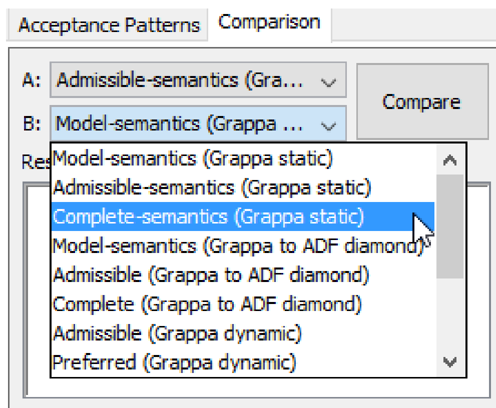
Figure 5.18: selection of a result to display

5.4.1 Comparison of evaluation methods

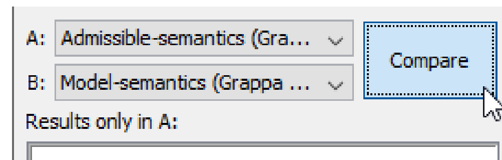
Sometimes it is convenient to compare two different evaluation methods resp. semantics. For this purpose, the comparison section is provided. Similar as in the evaluation panel, two evaluation methods can be chosen as shown in Fig. 5.19a.

By clicking the button “Compare” the selected evaluation methods are executed. During execution the button changes from “Compare” to “Stop”. The button “Stop” can be used to interrupt the execution.

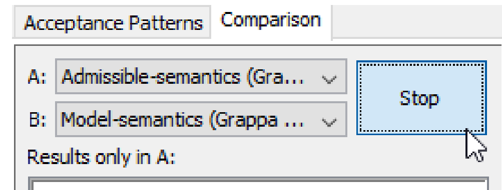
If the comparison was successful, the results are displayed in three lists as shown in Fig. 5.20: In the upper list – labeled “Results only in A” – interpretations are displayed



(a) choosing the evaluations for comparison



(b) starting comparison



(c) button changes from “Compare” to “Stop” during execution

Figure 5.19: comparison

which are only a result of the first selected evaluation method. In the lower list – labeled “Results only in B” – interpretations are shown which are only a result of the second selected evaluation method. The list in the middle – labeled “Results in A and B” – shows those results which are shared by both selected evaluation methods.

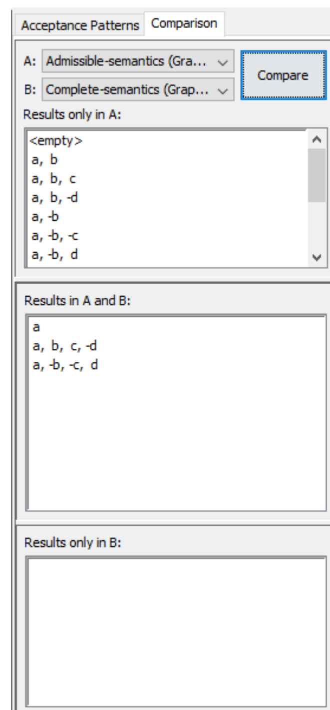


Figure 5.20: result of a comparison

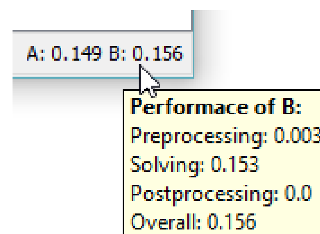


Figure 5.21: runtime of an evaluation

After executing an evaluation method, the runtime of the evaluation is shown in the lower right corner of the status bar. By moving the mouse cursor above this entry further information is displayed in a little window as shown in Fig. 5.21.

- The entry “Preprocessing” shows the required time to generate the encoding for the external solver. This can also comprise an external solver call, e. g. if the solver is necessary to convert a GRAPPA-instance to ADF.
- The entry “Solving” shows the required time of the external solver to process the encoding.
- The entry “Postprocessing” shows the required time of GrappaVis to process the result from the external solver.
- Finally, the entry “Overall” shows the whole runtime of the evaluation.

5.5 Syntax of the GRAPPA pattern language for GrappaVis

As described in the Section 5.3 the acceptance patterns for a GRAPPA-instance have to be imported. This section describes how acceptance patterns have to be specified for GrappaVis.

5.5.1 Overview

The basic part of the specification of an acceptance pattern are *definitions*. There are two types of definitions which are distinguished by the assignment-operator, either “`:=`” or “`:`”. The colon “`:`” indicates that the definition represents an acceptance pattern. On the other hand, “`:=`” defines an “additive expression”. This is not an acceptance pattern itself, but can be used within an acceptance pattern. An exact definition is given in Section 5.5.3.

The conversion of the pattern language of GRAPPA to the syntax used by GrappaVis is straight forward. Only the function-symbols of the pattern language are replaced by an ASCII-abstraction as shown in Fig. 5.22.

A feature of the syntax of GrappaVis is the possibility to define variables which can be used in acceptance patterns.

Example 5.1. Consider the acceptance pattern

$$\#_t(+) - \#(+) = 0 \land \#(-) = 0$$

There is not just only one way to encode this pattern for GrappaVis. The straight forward approach is shown in Listing 5.1.

function	replaced by
$\#$	`#`
$\#_t$	`#_t`
min	`min`
min_t	`min_t`
max	`max`
max_t	`max_t`
sum	`sum`
sum_t	`sum_t`
$count$	`count`
$count_t$	`count_t`

operator	replaced by
$\neg$	`NOT`
$\wedge$	`AND`
$\vee$	`OR`
$\otimes$	`XOR`
$<$	`<`
$\leq$	`<=`
$=$	`==`
$\neq$	`!=`
$\geq$	`>=`
$>$	`>`

Figure 5.22: translation of the pattern language

```
p1: #_t(+) - #(+) == 0 AND #(-) == 0;
```

Listing 5.1: encoding of the acceptance pattern of Example 5.1

But there is also the possibility to split the pattern into some useful parts by using variables as shown in Listing 5.2. The acceptance pattern in Line 1 is equivalent to the pattern in Listing 5.1. The advantage of this approach is, that the new variables `a1`, `a2` can be used to define other patterns as demonstrated in Line 5.

```
p1: a1 == 0 AND a2 == 0;
a1:= #_t(+) - #(+);
a2:= #(-);

p2: a2 OR min() > 0;
```

Listing 5.2: alternative encoding of the acceptance pattern of Example 5.1

△

5.5.2 Comments

It is also possible to specify comments. Comments are started by "%". Everything afterwards, within the same line is ignored by GrappaVis. An example is shown in Listing 5.3.

```
% all edges labeled with '+' must be active
% and there may be no acitve edge labeled with '-'
p1: #_t(+) - #(+) == 0 AND #(-) == 0;
```

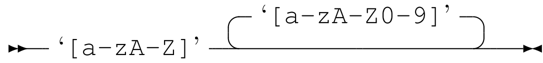
Listing 5.3: example of a comment

5.5.3 Syntax

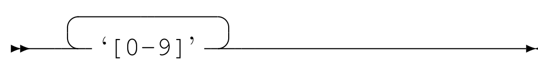
In this section the exact definition of the syntax is presented.

The representation of variables and numbers is specified by the following definitions.

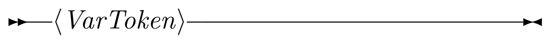
VarToken:



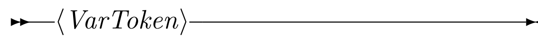
NumToken:



AddVariable:

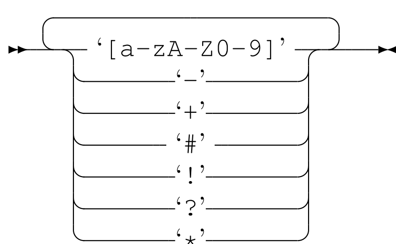


Variable:

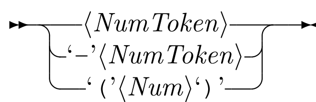


The difference between an *AddVariable* and a *Variable* is that the first may only occur within a functional combination while the latter replaces a whole acceptance pattern.

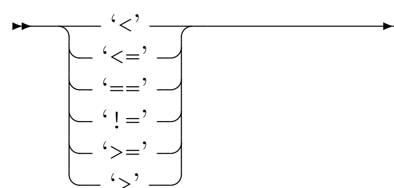
EdgeLabelToken:



Num:

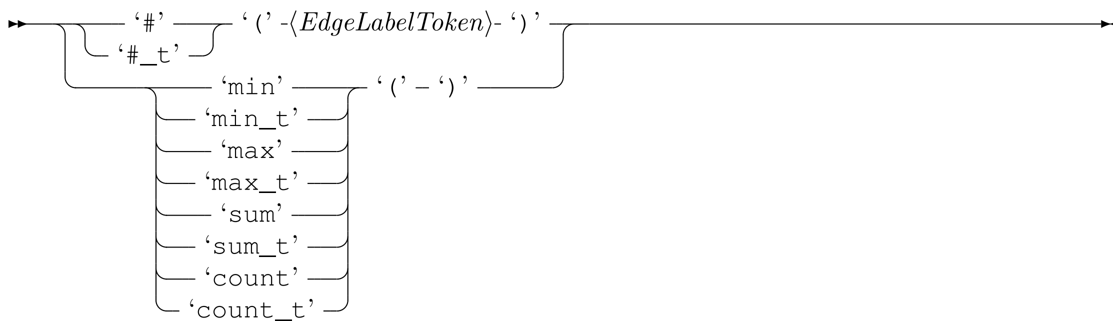


Comparator*:

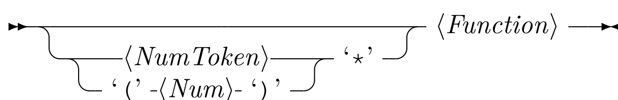


Starting from here the rules to build an acceptance pattern are presented.

Function:

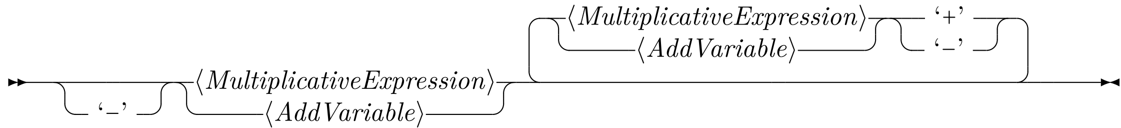


MultiplicativeExpression:

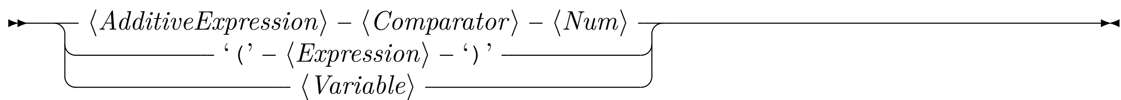


The syntax of *MultiplicativeExpression* guarantees that negative coefficients are always enclosed by brackets.

AdditiveExpression:



RelationalExpression:



ConditionalNotExpression[1]**:**

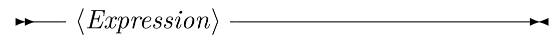
ConditionalAndExpression[1]**:**

ConditionalOrExpression[1]**:**

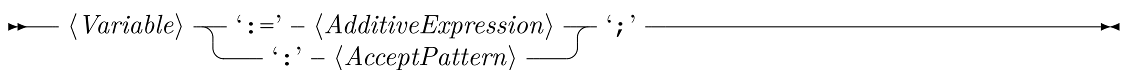
ConditionalXOrExpression[1]**:**

Expression*:

AcceptPattern:

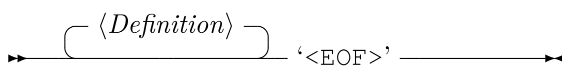


Definition:



The reason why “`:=`” is used for *AdditveExpressions* and “`:`” for *AcceptPatterns* is to make the productions easily distinguishable.

ExpressionList:

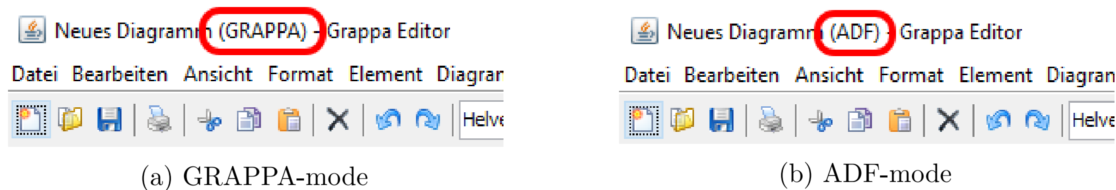


For parsing every production – not marked with “*” – is represented by a corresponding class within GrappaVis. These classes are used to represent the syntax-tree. Productions with “*” are omitted, i. e. children of them are added directly to the parent. Productions marked with “^{1}” are represented in the syntax-tree only if they have more than one child. This keeps the syntax-tree more manageable, because unnecessary productions are removed. The *ConditionalNotExpression* is only kept in the syntax-tree if there is a “NOT” at the corresponding position in the acceptance pattern to parse.

5.6 Using GrappaVis for ADFs

GrappaVis is also able to specify and evaluate ADF-instances. The handling is basically the same as for GRAPPA-instances with only minor differences.

After starting GrappaVis *GRAPPA-mode* is preselected. The current mode of GrappaVis is always displayed in the title of the program, i. e. either “(GRAPPA)” or “(ADF)” is shown as part of the title as depicted in Fig. 5.23a resp. Fig. 5.23b.



(a) GRAPPA-mode (b) ADF-mode

Figure 5.23: editing modes of GrappaVis

For an existing instance it is not possible to change the mode. If a new instance should be generated a window is shown – Fig. 5.24 – where the user can choose if he wants to specify a GRAPPA or an ADF-instance. Depending on the selected mode GrappaVis offers the appropriate evaluation-methods in the evaluation-toolbox, i. e. no evaluation methods for GRAPPA are offered for ADF-instances and the other way round.

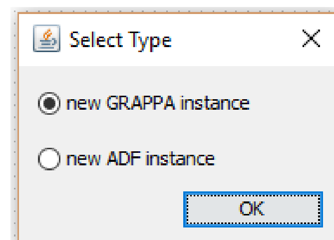


Figure 5.24: select the type to specify

The acceptance conditions for ADFs must be in the format, which is used by DIAMOND, i. e.

```
ac(<node>,<acceptance condition>).
```

where <node> specifies the id of the node where the acceptance condition given in <acceptance condition> should be applied.

The assignment of the acceptance condition to a node is done by the specification of the acceptance condition itself, hence no further assignment within GrappaVis is necessary.

Some examples for acceptance conditions are given in Listing 5.4:

```
1 ac(a,and(or(b,a),c)).
2 ac(b,or(d,and(c,a))).
3 ac(c,c(v)).
4 ac(d,c(f)).
```

Listing 5.4: acceptance conditions for ADF

The expressions c(f) resp. c(v) in the Lines 3 and 4 represent the constants *falsum* resp. *verum*. For a more elaborate description of the syntax consult [ES13].

5.7 Configuration of evaluation methods

To evaluate a given GRAPPA-instance, external programs need to be called and those programs may differ from system to system, or the start parameters may be different, especially if GrappaVis is executed on different operating systems. To keep the configuration of these calls flexible all necessary options can be expressed in the file *semantics.properties*.

In Listing 5.5 an example is given. In Line 1 the number of different evaluation methods, which are configured in this file, has to be defined.

```
1 no=1
2
3 Name0=Model-semantics (Grappa static)
4 DelFiles0=1
5 Option0=GrappaStaticEncoding
6 PreRun0=encodings/clingo -n0 %filename% encodings/basicDefs.lp
    ↪ encodings/toADF.lp
7 Run0=encodings/clingo -n0 %filename% encodings/basicDefs.lp
    ↪ encodings/model.lp
```

Listing 5.5: evaluation method configuration

For every configuration options are available, always followed by the id of the current configuration:

Name The string defined here is displayed in the drop-down list for selecting an evaluation method.

DelFiles The program writes temporary files. If this setting is ≥ 1 GrappaVis deletes those files when the solver finishes.

`Run` Here the command-line to execute a program can be defined. In most cases GrappaVis has to export a file which is an input for the external tool. The name of this temporary file is specified by GrappaVis and therefore can not be defined statically in the configuration file. To let GrappaVis know at which position the filename should be inserted into the command-line, the placeholder "`%filename%`" is used.

`PreRun` This setting is only used by the modes *GrappaToAdfDiamond* and *AdfDynamic*. Apart from that the function of this parameter is the same as for `Run`.

`Option` Here the evaluation method is defined. Possible values are:

`GrappaStaticEncoding` This option handles all static semantics for GRAPPA-instances. Which semantics is executed depends on the files which are passed to the ASP-Solver.

`GrappaDynamicEncodingAdm`

`GrappaDynamicEncodingPref`

`GrappaDynamicEncodingComp`

The three options above handle the corresponding semantics – admissible, preferred and complete – for the dynamic encoding approach.

`GrappaToAdfDiamond` With this option the GRAPPA-instance is converted into an ADF-instance. For this the parameter `PreRun` – in Line 6 of Listing 5.5 – is necessary, because the conversion again uses the ASP-solver. Which semantics is applied is determined by a parameter[1] which is passed to the DIAMOND solver within the parameter `Run` – in Line 7.

`AdfDiamond` This mode uses the DIAMOND-solver to evaluate an ADF-instance. Which semantics is applied is determined by a parameter which is passed to the DIAMOND solver.

`AdfToGrappaDynamicAdm`

`AdfToGrappaDynamicPref`

`AdfToGrappaStatic`

The three modes above convert an ADF-instance to a GRAPPA-instance and then exhibit the same functionality as the corresponding modes for GRAPPA, i. e. *GrappaDynamicEncodingAdm*, *GrappaDynamicEncodingPref*, *GrappaStaticEncoding*

[1] Refer to the DIAMOND manual [ES13] for parameter description.

5.8 Design decisions

5.8.1 Used software

GrappaVis is written in Java. The main reason for this decision is the fact that a very powerful and publicly available[2] graph-drawing framework, namely *JGraphX* [JGr16], is available for Java. Furthermore, *JGraphX* comes with an included graph editor, which is the base of GrappaVis and has been adapted and extended to meet the demands of editing and evaluating GRAPPA-instances.

To handle saving and loading files, the package *XStream* [CW16] was used. It provides an easy approach to (de)serialize objects.

The parser – for e. g. reading the input language of GRAPPA – is based on the *JavaCC* [Mic16] framework.

For parsing command-line parameters the *Apache Commons CLI* [Fou16] was used.

5.8.2 Regarding an editor for acceptance patterns

Although support for editing acceptance patterns directly within GrappaVis would have been a convenient feature, a simple textbox would not have sufficed because more advanced features of an ordinary text-editor like copy/ paste, undo/redo functions would have been missing. Moreover, it was not the goal to implement a (new) full-fledged text-editor. Therefore, every user can use his preferred text-editor to specify the acceptance patterns which then can be imported into GrappaVis.

[2]BSD-license

CHAPTER 6

Experimental evaluation

In this chapter results of testing and evaluating the encodings are presented. Because for GRAPPA there are no solvers available yet, existing solvers for ADFs were used to cross-check the results. To this end conversion-routines were implemented to convert ADF to GRAPPA – and the other way round – which are described in this chapter. Moreover, a small command-line program was implemented to easily generate random GRAPPA-instances. The parameters and usage of the program are also presented in this chapter.

6.1 Conversion from GRAPPA to ADF

A conversion from GRAPPA to ADF-instances was implemented. This conversion is based on the procedure described in [BW14]:

Definition 6.1. For a LAG $G = (S, E, L, \lambda, \alpha)$ define its associated ADF A_G as (S, E, C_G) where

$$C_G(s) = \bigvee_{T \subseteq par_E(s) : \alpha(s)(m_s^T) = t} \Big(\bigwedge_{r \in T} r \wedge \bigwedge_{r \in par_E(s) \setminus T} \neg r \Big).$$

To compute $C_G(s)$ for every $s \in S$ the encodings developed for the static encoding in Chapter 3 is used with some modifications.

The input-file is basically the same as for the static encoding described in Section 3.1 with the only difference that a new atom **aktNode(s)** is added to let the program know that the acceptance pattern is computed for node s. The computation of $C_G(s)$ works only on a "subset" of the GRAPPA-instance, i. e. only the acceptance pattern of the s and the parent nodes of s are required. Therefore, the input-file can be stripped off the atoms

- **s(X)** where $X \notin \{s\} \cup par_E(s)$
- **e(X,Y)** where $Y \neq s$

Because the input is basically the same for the static encoding the basic definitions, described in Section 3.3, can be used as usual. Only a few rules have to be added to retrieve the desired result, namely the set $\mathcal{T} = \left\{T \mid T \subseteq par_E(s) : \alpha(s)(m_s^T) = t\right\}$.

To retrieve $\mathcal{T}$ all possible subsets of parents of s are guessed in Line 3 of Listing 6.1. In Line 5 all interpretations T are removed for which the acceptance pattern of s evaluates to false, i. e. atom **nomodel** is derived for node s.

So every answer-set found for this program corresponds to a $T \in \mathcal{T}$, where the atom **in(0,r)** corresponds to $r \in T$ and **out(0,r)** corresponds to $r \in par_E(s) \setminus T$.

```
1 guess(0).
2
3 in(0,S) | out(0,S) :- s(S,_), e(S,X,_), aktNode(X).
4
5 :- aktNode(X), s(X,P), nomodel(0,P).
6
7 ok(0) :- #false.
```

Listing 6.1: compute $C_G(s)$ (from file *toADF.lp*)

Example 6.1. This example is taken from [BW14]. Let $S = \{a, b, c, d\}$ a LAG and $L = \{+, \text{-}\}$. The graph in Fig. 6.1 shows the labels of each link. The acceptance pattern for all nodes is

$$\#_t(+) - \#(+) = 0 \wedge \#(\text{-}) = 0$$

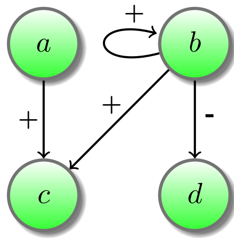


Figure 6.1: GRAPPA-instance of Example 6.1

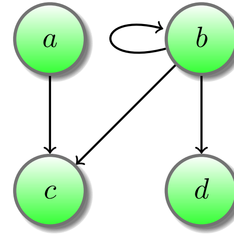


Figure 6.2: resulting ADF of Example 6.1

The acceptance conditions for the four nodes are computed like this:

- node a: $par_E(a) = \emptyset$, the ASP-program yields one answer-set T with neither **in** nor **out** atoms. $\Rightarrow T_1 = \emptyset, \mathcal{T} = \{T_1\}$

$$
\begin{aligned}
C_G(a) &= \bigvee_{T\in\mathcal{T}} \Big(\bigwedge_{r\in T} r \wedge \bigwedge_{r\in par_E(a)\setminus T} \neg r \Big) \\
&= \bigwedge_{r\in T_1} r \wedge \bigwedge_{r\in par_E(a)\setminus T_1} \neg r \\
&= \bigwedge_{r\in\emptyset} r \wedge \bigwedge_{r\in\emptyset\setminus\emptyset} \neg r \\
&= \top \wedge \top \\
&= \top
\end{aligned}
$$

- node b: $par_E(b) = \{b\}$, the ASP-program yields one answer-set including **in(b)** $\Rightarrow T_1 = \{b\}, \mathcal{T} = \{T_1\}$

$$
\begin{aligned}
C_G(b) &= \bigvee_{T\in\mathcal{T}} \Big(\bigwedge_{r\in T} r \wedge \bigwedge_{r\in par_E(b)\setminus T} \neg r \Big) \\
&= \bigwedge_{r\in\{b\}} r \wedge \bigwedge_{r\in\{b\}\setminus\{b\}} \neg r \\
&= b \wedge \top \\
&= b
\end{aligned}
$$

- node c: $par_E(c) = \{a, b\}$, the ASP-program yields one answer-set including **in(a)** and **in(b)** $\Rightarrow T_1 = \{a, b\}, \mathcal{T} = \{T_1\}$

$$
\begin{aligned}
C_G(c) &= \bigvee_{T\in\mathcal{T}} \Big(\bigwedge_{r\in T} r \wedge \bigwedge_{r\in par_E(c)\setminus T} \neg r \Big) \\
&= \bigwedge_{r\in\{a,b\}} r \wedge \bigwedge_{r\in\{a,b\}\setminus\{a,b\}} \neg r \\
&= a \wedge b \wedge \top \\
&= a \wedge b
\end{aligned}
$$

- node d: $par_E(d) = \{b\}$, the ASP-program yields one answer-set including **out(b)** $\Rightarrow T_1 = \emptyset, \mathcal{T} = \{T_1\}$

$$
\begin{aligned}
C_G(d) &= \bigvee_{T\in\mathcal{T}} \Big(\bigwedge_{r\in T} r \wedge \bigwedge_{r\in par_E(d)\setminus T} \neg r \Big) \\
&= \bigwedge_{r\in\emptyset} r \wedge \bigwedge_{r\in\{b\}\setminus\emptyset} \neg r \\
&= \top \wedge \neg b \\
&= \neg b
\end{aligned}
$$

This yields the ADF-instance shown in Fig. 6.2.

△

6.2 Conversion from ADF to GRAPPA

A conversion from ADF to GRAPPA was implemented as well. This is easier than the other way round, because only two steps have to be performed – also described in [BW14]:

1. Label every edge from the ADF-instance with the label of the source node.
2. To obtain the acceptance pattern for GRAPPA replace every occurrence of a node n within an acceptance pattern of ADF by $\#(n) = 1$.

Example 6.2. Let $D = (S, L, C)$ an ADF, $S = \{a, b, c\}$, $L = \{(a, c), (b, c)\}$, $\phi_c = a \wedge \neg b$ an acceptance pattern for node c.

The corresponding acceptance pattern of node c is

$$\#(a) = 1 \wedge \neg\#(b) = 1,$$

where a, b are edgelabels, in the resulting GRAPPA-instance.

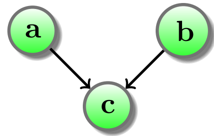


Figure 6.3: ADF instance of Example 6.2

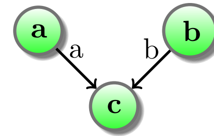


Figure 6.4: conversion result of Example 6.2

△

6.3 GRAPPA-instance generation

To generate random GRAPPA-instances a command-line tool was implemented, which is called *gen.jar*.

The following parameters must be provided for execution:

- `-cnt`: specifies how much GRAPPA-instances should be generated
- `-n, -nodes`: specifies the number of nodes the GRAPPA-instances should comprise
- `-e, -edges`: specifies the number of edges the GRAPPA-instances should comprise
- `-f, -file`: specifies the name of the file(s) to generate. To the name given here a number is appended by the generator to distinguish the different files.

For example a call of the generator looks like this:

```
java -jar gen.jar -cnt 5 -n 10 -e 20 -f grappa/inst10_20
```

The given call will generate five GRAPPA-instances with ten nodes, twenty edges and write them into the directory "grappa" with the names *inst10_1.xml ... inst10_5.xml*

Between which nodes the edges are placed is determined randomly, as well as the acceptance patterns of the statements.

6.4 Performance

Performance was not the main concern on developing the encodings for GRAPPA, because at the moment there are not any other GRAPPA-solvers to compare them to anyway. Even so some comparisons were done between the static and dynamic approach and also between the presented GRAPPA-encodings and existing ADF-solvers by converting GRAPPA-instances to ADF-instances and the other way round.

The tests were executed on a *Thinkpad Yoga* with an i7-4500U processor 1.80GHz and 8 GB RAM. This is not a machine for benchmarking, but these performance tests should only give an impression how the different encodings and solvers perform.

6.4.1 Dynamic vs. static encoding

As expected the static encoding is much slower than the dynamic encoding. The reason is obvious: In the static case the encoding needs to take care about parsing patterns and is designed to work on every possible instance. The dynamic encoding does not need to consider that. This runtime difference is significant. On a random instance of 10 nodes and 20 edges the static encoding for the admissible semantics was interrupted after 15 minutes whereas the dynamic encoding finished on the same instance within 35 seconds.

Still the dynamic encoding in this form soon reaches its limits. On an instance with 20 nodes and 40 edges the solver needs more than 8 minutes to compute the admissible interpretations. At this point an ASP preprocessing tool for decomposing rule-bodies, which was originally introduced in [MW12] and further extended in [Bic15], was used to improve the performance. With this tool the same instance was solved within about 30 seconds. Actually the solving takes about 20 seconds and the remaining time is used by GrappaVis for post-processing the results, because there are already over 250 000 admissible interpretations. The runtime of the tool itself is already included in the 20 seconds. For comparison: The instance with the 10 nodes is solved in less than 1 second with the decomposition tool, in contrast to 35 seconds without. This configuration is referred as *optimized dynamic encoding*.

But the amount of resulting interpretations for a semantics also causes another problem, namely that for even larger instances – therefore potentially exponential more admissible interpretations – GrappaVis is not longer able to handle them. Some tests were done on instances with up to 50 nodes. Although they were solvable, even if it took some minutes, GrappaVis crashed due to memory shortage.

To get an impression of the performance of the different encodings, in Table 6.1 runtimes of the encodings are listed. The presented times are results of running 20 different instances and are denoted “$< t$ sec” indicating that all instances were processed in less

than t seconds. The times given in column “opt” are resulting from the dynamic encoding togther with the preprocessing tool from [Bic15].

	static	dynamic	opt
admissible	> 10 min	< 35 sec	< 2 sec
complete	> 10 min	> 10 min	< 4 sec
preferred	(not impl.)	> 10 min	< 3 sec

Table 6.1: comparison of runtimes for instances with 10 nodes

In Table 6.2 a comparison of the performance of the optimized dynamic encoding is given for different instance sizes. The instance size is given by two numbers, e. g. 10/20, where the first gives the number of nodes (10) and the second the number of edges (20). The presented times are results of running 20 different instances and are denoted “$< t$ sec (X/20)” indicating that X instances – out of 20 instances – were processed in less than t seconds.

	10/20	20/50	50/120
admissible	< 2 sec (20/20)	< 30 sec (20/20)	(out of mem)
complete	< 4 sec (20/20)	< 7 sec (18/20) < 17 sec (20/20)	< 30 sec (17/20) < 52 sec (19/20) < 135 sec (20/20)
preferred	< 3 sec (20/20)	< 6 sec (19/20) < 34 sec (20/20)	> 10 min

Table 6.2: runtimes for different instance sizes

6.4.2 GRAPPA vs. ADF

In Table 6.3 runtimes of different solvers are given as a result of processing 20 different ADF-instances. Again the smaller sign “<” indicates that all instances were processed in less than the given time. There were no significant runtime differences between the different instances.

DIAMOND is a solver for ADF-instances [ES13] and for “GRAPPA dyn. opt” the ADF-instance is converted to a GRAPPA-instance– as described in Section 6.2 – and then evaluated by the optimized dynamic encoding.

	DIAMOND	GRAPPA dyn. opt
admissible	< 5 sec	< 15 sec
complete	< 5 sec	< 40 sec
preferred	< 6 sec	< 30 sec

Table 6.3: comparison of runtimes for ADF-instances with 10 nodes

Obviously it is not the best idea to solve ADF-instances by converting them to GRAPPA-instances instead of using a native ADF-solver. The reason why the GRAPPA approach here is much slower, is – most probably – that the conversion of ADF to GRAPPA inherently increases the complexity of the instance. Moreover, GRAPPA is the more general framework and therefore the encoding is more complex than the encoding of an ADF-instance.

Similar as in Table 6.3, in Table 6.4 runtimes of different solvers are given as a result of processing 20 different GRAPPA-instances. To process the GRAPPA-instances with DIAMOND the instances are converted to ADF as described in Section 6.1.

	DIAMOND	GRAPPA dyn. opt
admissible	< 5 sec	< 3 sec
complete	< 3 sec	< 4 sec
preferred	< 3 sec	< 3 sec

Table 6.4: comparison of runtimes for GRAPPA-instances with 10 nodes

In this case the runtime differences are not as significant as before, when ADF was converted to GRAPPA. Still it is surprising that the evaluation with DIAMOND is that fast, because the stated runtime includes also the required conversion time. Even more astonishing if considered that the conversion from GRAPPA to ADF requires an ASP-solver call for each node.

CHAPTER 7

Conclusion and future work

Dung introduced AFs in his landmark work [Dun95], which inspired many generalizations of AFs and among them, GRAPPA – presented in [BW14] – is one of the most recent and most general frameworks. But so far GRAPPA was "only" a theoretical concept. Therefore the main goal of this work is to provide tools to make GRAPPA available for practical use.

To this end a *static* ASP-encoding for GRAPPA has been presented, i. e. with this encoding GRAPPA-instances can be evaluated over different semantics. This encoding is static in a way, that it does not change for different instances and supports model, admissible and complete semantics. To overcome side-effects in the encoding, which arise on using ASP-aggregates together with the saturation methodology, reformulations for default ASP-aggregates have been developed and applied to the encodings. These reformulations ensure that the indented behavior of an aggregate is not altered unintentionally by saturation.

Because the *static* encoding approach is limited in terms of the possibilities to formulate more complex semantics, a new *dynamic* approach has been introduced. This approach is exploiting two things: The encoding is written for each instance individually, therefore some preprocessing of the instance can be done. Moreover the encoding utilizes a methodology to encode an NP-complete problem into one single rule-body. Thus not only an encoding for admissible and complete semantics, but also for preferred semantics has been provided, what has not been possible for the *static* encoding.

Furthermore GrappaVis was developed and has been presented as a graphical tool to specify GRAPPA-instances and further to evaluate them. For evaluation GrappaVis performs the necessary processing of the specified GRAPPA-instance for a selected evaluation method.

But basically GrappaVis is a graph-editor and a tutorial on how to use GrappaVis

- to draw a graph,
- to assign acceptance patterns to nodes of the graph,
- to execute evaluations methods, i. e. different semantics provided by different encodings or other tools,
- to compare different evaluation methods on a given GRAPPA-instance

has been presented.

Finally, to verify the newly developed encodings, conversions from ADF to GRAPPA, and the other way round, have been implemented to enable the use of existing solvers for ADF. Moreover, the usage of a tool to generate GRAPPA-instances has been explained. Also some performance comparisons have been done, which have shown clearly, that the most promising approach for evaluating GRAPPA-instances is the dynamic approach together with a preprocessing-tool, which optimizes the encoding. Surprisingly the solvers for ADF are performing very well, too, i. e. even though the instances have to be converted from GRAPPA to ADF the overall performance is not much slower.

In general, further research towards performance optimization can be done. Especially the preprocessing of the dynamic encoding has been implemented in a straight forward approach. That means that there are some points which could be reconsidered if there is a more sophisticated way to implement them, to improve the overall performance. Moreover, the usage of ADF solvers for GRAPPA-instances can be examined on a broader base, i. e. with more and larger instances, to get an impression how the performance changes in comparison to the "native" optimized dynamic encoding. Also the conversion routine from GRAPPA to ADF could be a starting point, because the developed conversion is based on the static encoding. Maybe a dynamic approach for the conversion can speed up the conversion and therefore also the overall performance.

Another point for future research is the implementation of more semantics, as for example the stable model semantics or the grounded semantics.

Regarding GrappaVis there are many things which could be improved or changed. For example a full-fledged editor could be integrated, which would simplify the handling of acceptance patterns. Moreover, the handling of the results of an evaluation could be improved. At the moment GrappaVis has problems to handle evaluations which yield more than 250 000 results[1]. Moreover, the handling of ADFs could be improved, because at the moment the support of ADFs is implemented only on a rudimentary basis.

[1] Keep in mind that for an instance with n nodes, evaluated over the admissible encoding, there are 3^n results in the worst case.

Index

ADF, 2, 3, 8–11, 30, 65, 75, 79–81, 83–89, 92
- admissible, 10
- characteristic operator, 10
- complete, 10
- model, 10
- preferred, 10

AF, 8
- acceptable, 8
- admissible, 8
- attack, 8
- complete, 8
- conflict-free, 8
- preferred, 8

ASP, IX, XI, 2, 3, 5, 14–16, 18–24, 27, 29–31, 33, 44, 55, 57–59, 81, 84, 85, 87, 89, 91
- answer-set, 20
- atom
  - ground, 18
  - non-ground, 18
- body, 15
- bounded arity, 31
- constraint, 16
- default negation, 19
- derive, 15
- disjunctive program, 30
- domain predicate, 25
- dynamic encoding, 31
- fact, 16
- grounding, 18
- head, 15
- interpretation, 20
- interval, 18
- model, 20
  - candidate, 20
  - minimal, 20
- normal program, 30
- program, 15
  - disjunctive, 16
  - ground, 18
  - non-ground, 18
  - normal, 16
- reasoning
  - brave, 29
  - cautious, 29
- reduct, 19
- rule, 15
  - disjunctive, 15
  - fires, 15
  - ground, 18
  - non-ground, 18
  - normal, 16
  - positive, 16
- safe variable, 18
- saturation, 23
- stable model, 20
- stable model semantics, 19
- static encoding, 31
- strong negation, 19

characteristic operator
- GRAPPA, 14
- LAG, 11

complexity
- combined complexity, 31
- data complexity, 30

functional combination, 34

GRAPPA, VII, IX, XI, 2, 3, 8, 11, 29–31, 36–39, 48, 57, 58, 65, 75, 79, 81–83, 86–89, 91, 92
  acceptance pattern, 13
  basic acceptance pattern, 13
  characteristic operator, 14
  instance, 12
  reasoning
    brave, 29
    cautious, 29
  satisfaction relation, 14
  term, 13
GrappaVis, IX, XI, 3, 55, 65, 66, 68, 75, 76, 79–82, 87, 91, 92
  acceptance pattern panel, 67
  acceptance pattern section, 67
  comparison section, 67
  default label, 67
  drag and drop, 65
    drag-source, 65
    drop-position, 65
    drop-target, 65
  drawing area, 66
  editing mode
    edge, 69
    node, 69
  evaluation panel, 67
  evaluation toolbox, 67
  logging area, 66
  overview panel, 66
  performance
    overall, 75
    Postprocessing, 75
    preprocessing, 75
    solving, 75
  result entry
    <empty>, 72
    <none>, 72
  status bar, 66
  template panel, 66

LAG, 11
  characteristic operator, 11
  semantics, 12
Link
  active, 11

meet operator, 7, 10

oracle, 30

polynomial hierarchy, 29
predicate
  accept, 43, 51, 52
  activeedge, 45, 62
  activelabelcount, 36, 39, 45, 48, 49
  aktNode, 83
  alcount, 45, 47, 48
  alcount_max, 49
  and, 34, 36, 38, 39, 41
  arg, 56
  ass, 56, 57, 63
  ass2, 62, 63
  b, 23
  basicpattern, 34, 36–39, 41–44
  cnt, 25
  cntActLabel, 46
  distactivelabel, 44, 47, 48
  distinctactivelabel, 37, 39, 47, 49
  distinctlabel, 37, 39
  distintactivelabel, 44
  distlabel, 44, 45, 48
  dom, 25, 26
  dom_cntActLabel, 46
  e, 21, 34, 37, 84
  eq, 42
  false, 42
  g, 23
  geq, 42
  gt, 42
  guess, 22, 45, 51, 52, 54
  in, 24, 25, 27, 28, 40, 51, 52, 84, 85
  intv, 18, 19
  ismodel, 43
  label_max, 48
  label_min, 48

labelcount, 37, 39
lcount, 44
leq, 42, 56, 57
lhspat, 42, 50
lhspat_dom, 50
lt, 42
maxactivelabel, 38, 39, 46, 49
maxalabel, 46–48
maxDom, 38, 49, 71
maxlabel, 38, 39
maxtlabel, 44
maxVal, 46
min, 25
minactivelabel, 38, 39, 45, 46, 49
minalabel, 45, 46, 48, 49
minalablel_dom, 46
minDom, 38, 71
minlabel, 38, 39
mintlabel, 44
mterm, 48–50
mterm_max, 48–50
neg, 36, 38, 39, 41
neq, 42
nomodel, 43, 84
notaccept, 43, 52
ok, 24, 26, 28, 29, 45, 52, 53
or, 34, 36, 38, 39, 41
out, 25, 27, 28, 40, 51, 52, 84, 85
p, 26
pattern, 41–43
r, 23
s, 21, 28, 34, 39, 41, 84
sat, 56, 57, 59–61, 63
sat2, 63
saturate, 63
sum, 25
sumactivelabel, 39, 47, 49
sumalabel, 47
sumlabel, 39
sumtlabel, 44
term, 34, 37, 39
true, 42
undec, 41, 51, 52, 61, 62
undef, 54
unsat, 56, 57, 59–61, 63
unsat2, 63
valid, 22
validColoring, 22
xor, 34, 36, 38, 39, 41

syntax
AcceptPattern, 78
accPattern, 36
AdditiveExpression, 78
AddVariable, 77
aggregate, 16
aggregate atom, 16
atom, 16
binary atom, 16
body, 15
Comparator, 77
comparator, 17
ConditionalAndExpression, 78
ConditionalNotExpression, 78
ConditionalOrExpression, 78
ConditionalXOrExpression, 78
constant, 17
Definition, 78
EdgeLabelToken, 77
Expression, 78
ExpressionList, 78
Function, 77
head, 15
id, 35
Integer, 35
MultiplicativeExpression, 77
Num, 77
NumToken, 77
RelationalExpression, 78
rule, 15
s, 35
term, 17, 36
tuple, 16
Variable, 77
variable, 17
VarToken, 77

three-valued logic
completion, 8
extension, 7
information order
sets, 8
variables, 7
interpretation, 7
literal, 7
two-valued logic
boolean formula, 6
extension, 6
formula, 6
interpretation, 6
syntax, 5

Glossary

`ASCII` American Standard Code for Information Interchange 15, 75

`DIAMOND` *DIAlectical MOdels eNcoDing*; a system to evaluate ADF instances ([ES13]) 2, 79, 81, 88, 89

`e. g.` from Latin, abbreviation of *exempli gratia* ("for example") 2, 3, 5, 6, 17, 36, 37, 48, 75, 82, 88

`GRAPPA` GRaph-based Argument Processing with Patterns of Acceptance [BW14] VII, IX, XI, XIII, XIV, 2, 3, 8, 11–14, 29–31, 33, 34, 36–40, 48, 50–52, 54–59, 61, 65–67, 71–73, 75, 79–84, 86–89, 91, 92

`i. e.` from Latin, abbreviation of *id est* ("it is"); usually spoken as "that is" XI, 1, 2, 6, 9, 13, 15–17, 21, 22, 24, 30, 31, 34, 49, 51, 52, 54, 56–58, 61, 62, 65, 66, 69, 72, 79, 81, 83, 84, 91, 92

`iff` if and only if 6–8, 10, 12, 14, 51, 54, 62

`Potassco` Potsdam Answer Set Solving Collection [GKK^{+}11] 15

`resp.` respectively 2, 6, 29–31, 36, 42, 43, 48, 50, 51, 56–58, 60, 71, 73, 79, 80

`s. t.` such that 8, 20, 21, 29, 54, 56

`w. l. o. g.` without loss of generality 22

`w. r. t.` with respect to 2, 8, 20

Acronyms

ADF abstract dialectical framework XIII, 2, 3, 8–11, 30, 65, 75, 79–81, 83–89, 92

AF argumentation framework XI, XIII, 1, 2, 8, 9, 91

AI Artificial Intelligence XI, 1

ASP Answer Set Programming IX, XI, XIII, 2, 3, 5, 14–25, 27, 29–31, 33, 44, 51, 55, 57–59, 81, 84, 85, 87, 89, 91

LAG labeled argument graph 11, 12, 14, 83, 84

LHS left-hand-side 34, 36, 37

QBF quantified boolean formula 2

RHS right-hand-side 34, 36, 37

Bibliography

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic Programming and Negation: A Survey. *J. Log. Program.*, 19/20:9–71, 1994.

[AFG15] Mario Alviano, Wolfgang Faber, and Martin Gebser. Rewriting recursive aggregates in answer set programming: back to monotonicity. *TPLP*, 15(4-5):559–573, 2015.

[AP09] Leila Amgoud and Henri Prade. Using arguments for making and explaining decisions. *Artif. Intell.*, 173(3-4):413–436, 2009.

[BBD+12] Cristian E. Briguez, Maximiliano Celmo Budán, Cristhian A. D. Deagustini, Ana Gabriela Maguitman, Marcela Capobianco, and Guillermo Ricardo Simari. Towards an Argument-based Music Recommender System. In Bart Verheij, Stefan Szeider, and Stefan Woltran, editors, *Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10-12, 2012*, volume 245 of *Frontiers in Artificial Intelligence and Applications*, pages 83–90. IOS Press, 2012.

[BD07] Trevor J. M. Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. *Artif. Intell.*, 171(10-15):619–641, 2007.

[BH08] Philippe Besnard and Anthony Hunter. *Elements of Argumentation.* MIT Press, 2008.

[Bic15] Manuel Bichler. Optimizing Non-Ground Answer Set Programs via Rule Decomposition. Bachelor's thesis, Vienna University of Technology, 2015.

[Bid91] Nicole Bidoit. Negation in Rule-Based Database Languages: A Survey. *Theor. Comput. Sci.*, 78(1):3–83, 1991.

[BLS14] Maximiliano Celmo Budán, Mauro Javier Gómez Lucero, and Guillermo Ricardo Simari. An AIF-Based Labeled Argumentation Framework. In Christoph Beierle and Carlo Meghini, editors, *Foundations of Information and Knowledge Systems - 8th International Symposium, FoIKS 2014, Bordeaux, France, March 3-7, 2014. Proceedings*, volume 8367 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2014.

[BPW14] Gerhard Brewka, Sylwia Polberg, and Stefan Woltran. Generalizations of Dung Frameworks and Their Role in Formal Argumentation. *IEEE Intelligent Systems*, 29(1):30–38, 2014.

[BSE+13] Gerhard Brewka, Hannes Strass, Stefan Ellmauthaler, Johannes Peter Wallner, and Stefan Woltran. Abstract Dialectical Frameworks Revisited. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 803–809. IJCAI/AAAI, 2013.

[BW10] Gerhard Brewka and Stefan Woltran. Abstract Dialectical Frameworks. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*, pages 102–111. AAAI Press, 2010.

[BW14] Gerhard Brewka and Stefan Woltran. GRAPPA: A Semantical Framework for Graph-Based Argument Processing. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 153–158. IOS Press, 2014.

[CMS04] Carlos Iván Chesñevar, Ana Gabriela Maguitman, and Guillermo Ricardo Simari. A first approach to argument-based recommender systems based on defeasible logic programming. In James P. Delgrande and Torsten Schaub, editors, *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings*, pages 109–117, 2004.

[CMS07] Carlos Iván Chesñevar, Ana Gabriela Maguitman, and Guillermo Ricardo Simari. Recommender System Technologies based on Argumentation 1. In Ilias Maglogiannis, Kostas Karpouzis, Manolis Wallace, and John Soldatos, editors, *Emerging Artificial Intelligence Applications in Computer Engineering - Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, volume 160 of *Frontiers in Artificial Intelligence and Applications*, pages 50–73. IOS Press, 2007.

[CW16] XStream Committers and Joe Walnes. XStream. `http://x-stream.github.io/index.html`, 2016. [Online; accessed 10-January-2016].

[Dun95] Phan Minh Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artif. Intell.*, 77(2):321–358, 1995.

[DWW14] Martin Diller, Johannes Peter Wallner, and Stefan Woltran. Reasoning in Abstract Dialectical Frameworks Using Quantified Boolean Formulas. In Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti, editors, *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, volume 266 of *Frontiers in Artificial Intelligence and Applications*, pages 241–252. IOS Press, 2014.

[EFFW07] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-solving Using the DLV System. In Jack Minker, editor, *Logic-based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[EG95] Thomas Eiter and Georg Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[EIKP08] Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner, and Axel Polleres. Rules and Ontologies for the Semantic Web. In Cristina Baroglio, Piero A. Bonatti, Jan Maluszynski, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, volume 5224 of *Lecture Notes in Computer Science*, pages 1–53. Springer, 2008.

[EIP+06] Thomas Eiter, Giovambattista Ianni, Axel Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with Rules and Ontologies. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web, Second International Summer School 2006, Lisbon,*

Portugal, September 4-8, 2006, Tutorial Lectures, volume 4126 of *Lecture Notes in Computer Science*, pages 93–127. Springer, 2006.

[Ell12] Stefan Ellmauthaler. Abstract Dialectical Frameworks; properties, complexity, and implementation. Master's thesis, Vienna University of Technology, 2012. Diplomarbeit.

[ELS98] Thomas Eiter, Nicola Leone, and Domenico Saccà. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theor. Comput. Sci.*, 206(1-2):181–218, 1998.

[ES13] Stefan Ellmauthaler and Hannes Strass. The DIAMOND System for Argumentation: Preliminary Report. *CoRR*, abs/1312.6140, 2013.

[Fou16] Apache Software Foundation. Commons CLI. `https://commons.apache.org/proper/commons-cli/`, 2016. [Online; accessed 10-January-2016].

[FPL11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

[GKK+11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[GL88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[GL02] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation - The A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.

[GPW07] Thomas F. Gordon, Henry Prakken, and Douglas Walton. The Carneades model of argument and burden of proof. *Artif. Intell.*, 171(10-15):875–896, 2007.

[JGr16] JGraph Ltd. JGraphX. `https://github.com/jgraph/jgraphx`, 2016. [Online; accessed 10-January-2016].

[Joh90] David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161. 1990.

[KL94] Hans Kleine Büning and Theodor Lettmann. *Aussagenlogik - Deduktion und Algorithmen*. Leitfäden und Monographien der Informatik. Teubner, 1994.

[Kle09] Stephen C. Kleene. *Introduction to Metamathematics*. Ishi Press International, March 2009.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[Mic16] Sun Microsystems. JavaCC. `https://javacc.java.net/`, 2016. [Online; accessed 10-January-2016].

[MT99] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.

[MW12] Michael Morak and Stefan Woltran. Preprocessing of Complex Non-Ground Rules in Answer Set Programming. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 247–258. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[Nie99] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[PHR+11] Philippe Pasquier, Ramon Hollands, Iyad Rahwan, Frank Dignum, and Liz Sonenberg. An empirical study of interest-based negotiation. *Autonomous Agents and Multi-Agent Systems*, 22(2):249–288, 2011.

[PS01] Alessandro Provetti and Tran Cao Son, editors. *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, March 26-28, 2001*, 2001.

[RS09] Iyad Rahwan and Guillermo Ricardo Simari. *Argumentation in Artificial Intelligence*. Springer Verlag, 2009.

[SA15] Christian Strasser and G. Aldo Antonelli. Non-monotonic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2015 edition, 2015. `http://plato.stanford.edu/archives/fall2015/entries/logic-nonmonotonic/`.

[Sim11] Guillermo Ricardo Simari. A Brief Overview of Research in Argumentation Systems. In Salem Benferhat and John Grant, editors, *Scalable Uncertainty Management - 5th International Conference, SUM 2011, Dayton, OH, USA, October 10-13, 2011. Proceedings*, volume 6929 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2011.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 1–9. ACM, 1973.

[Sto76] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[SW14] Hannes Strass and Johannes Peter Wallner. Analyzing the Computational Complexity of Abstract Dialectical Frameworks via Approximation Fixpoint Theory. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, pages 101–110. AAAI Press, 2014.

[SW15] Hannes Strass and Johannes Peter Wallner. Analyzing the computational complexity of abstract dialectical frameworks via approximation fixpoint theory. *Artif. Intell.*, 226:34–74, 2015.

[vdWDM+11] Thomas L. van der Weide, Frank Dignum, John-Jules Ch. Meyer, Henry Prakken, and Gerard Vreeswijk. Multi-criteria argument selection in persuasion dialogues. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 921–928. IFAAMAS, 2011.