Unterschrift des Betreuers

MASTERARBEIT

On Debugging of Propositional Answer-Set Programs

Ausgeführt am

Institut für Informationssysteme
Abteilung für Wissensbasierte Systeme

der

Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Mag.rer.nat. Dr.techn. Hans Tompits

sowie der begleitenden Betreuung von

Univ.Ass. Dipl.-Ing. Dr.techn. Stefan Woltran

durch

Jörg Pührer, Bakk.techn.

Gentzgasse. 14/2/3, 1180 Wien

1. Februar 2008

Unterschrift des Studenten

Deutsche Zusammenfassung

Diese Masterarbeit beschäftigt sich mit Debugging in der Antwortmengen-Programmierung (engl., „answer-set programming - ASP“), einem Paradigma logischer Programmierung, das sich aufgrund seiner deklarativen Semantik in den letzten Jahren erhöhter Beliebtheit in der wissenschaftlichen Gemeinschaft erfreut. Ein wesentlicher Grund, warum sich ASP jedoch bisher nicht in größerem Maße als Problemlösungstechnik etablieren konnte, ist die mangelnde Verfügbarkeit von Programmier-Werkzeugen, welche die Arbeit mit ASP unterstützen. Insbesondere ist die Suche nach Fehlern in Antwortmengen-Programmen ein noch wenig erforschtes Gebiet, das höchste Aufmerksamkeit verdient.

Diese Arbeit führt zwei neue Techniken ein, die sich der Fehlersuche in Antwortmengen-Programmen widmen und sich dabei selbst der Antwortmengenprogrammierung bedienen. Eine der neuen Methoden basiert auf *ASP-Meta-Programmierung*, die andere auf einer sogenannten *Tagging-Technik*. Neben der Beschreibung der neuen Ansätze wird ein Überblick auf bestehende Publikationen zur Fehlersuche in Antwortmengenprogrammen gegeben.

Die Methode basierend auf Meta-Programmierung stellt den Hauptbeitrag dieser Arbeit dar. Sie beruht auf einer Transformation des potentiell fehlerhaften Programms Π in die Sprache eines Meta-Programms $\mathcal{D}_{\mathcal{M}}(\Pi)$. Die Kodierung erlaubt es, $\mathcal{D}_{\mathcal{M}}(\Pi)$ dazu heranzuziehen sämtliche Informationen über das ursprüngliche Programm Π zu extrahieren. Jede Antwortmenge des Meta-Programms trifft dabei Aussagen über das Verhalten von Π unter einer Interpretation I, die nicht zu den Antwortmengen von Π gehört. Umgekehrt existiert für jedes solche I eine entsprechende Antwortmenge des Meta-Programms. Der Kernansatz liegt nun darin, Gründe auszumachen, warum I nicht Antwortmenge von Π ist. Diese Information findet sich in den Antwortmengen des Meta-Programms, wobei zwischen vier verschiedenen Ursachen unterschieden wird:

- der Rumpf einer Regel ist erfüllt, aber kein Element des entsprechenden Regelkopfes;
- ein Atom ist in I enthalten, ohne Element des Kopfes einer anwendbaren Regeln zu sein;
- ein Integrity Constraint wird durch I verletzt; oder
- die Wahrheit eines Atoms aus I begründet sich nur durch sich selbst.

Die Methode erlaubt es komplexe Debugging-Abfragen durch Antwortmengenprogramme zu formulieren. Insbesondere kann die Menge an Informationen, die eine fehlerhafte Situation beschreiben, mithilfe gängiger ASP-Optimierungsverfahren auf relevante Anteile reduziert werden.

Bei der zweiten neuen Debugging-Methode handelt es sich um die Adaptierung einer Technik, die der Überführung von geordneten logischen Programmen in gewöhnliche logische

Programme dient. Durch die Einführung spezieller Atome, genannt *tags*, ist es möglich die Anwendbarkeit einzelner Regeln zu kontrollieren und verschiedene Eigenschaften über das fehlerhafte Programm aus den Antwortmengen einer Transformation des Programms abzulesen.

Ein Hauptmerkmal beider neuen Debuggingstrategien ist ihre Unabhängigkeit vom Algorithmus zur Berechnung der Antwortmengen. In dem nicht, wie in anderen Sparten der logischen Programmierung, dem Lösungsalgorithmus für die Fehlersuche gefolgt wird (vgl. Tracing in `PROLOG`), kann die deklarative Sicht auf Antwortmengen-Programme auch beim Debuggen beibehalten werden.

Im Rahmen dieser Arbeit wurde ein Prototyp einer Software zum Debuggen von Antwortmengenprogrammen implementiert. Dieses Werkzeug, genannt `spock`, unterstützt die Transformationen, die für die neu eingeführten Ansätze zur Fehlersuche in Antwortmengenprogrammen benötigt werden.

Im Folgenden wird ein Überblick über die einzelnen Kapitel der vorliegenden Masterarbeit gegeben:

Kapitel 1 gibt eine Einführung in die Prinzipien von ASP und in die allgemeinen Grundlagen der Fehlersuche in der Programmierung. Es widmet sich zunächst dem historischen sowie allgemeinen Kontext in dem Antwortmengenprogrammierung zu sehen ist. Nach einem Überblick über Debugging in verschiedenen Programmierparadigmen wird auf die grundsätzliche Problematik der Fehlersuche in Antwortmengenprogrammen eingegangen.

Im darauffolgenden Kapitel wird der formale Rahmen der Antwortmengenprogrammierung eingeführt; neben Programm-Syntax und Antwortmengen-Semantik werden wichtige Eigenschaften von Programmen und weitere Konventionen beschrieben, welche die theoretische Grundlage für die darauffolgenden Kapitel dieser Arbeit liefern.

In Kapitel 3 wird eine neue Debugging-Methode für Antwortmengenprogramme beschrieben. Zu einem fehlerhaften Programm, Π, wird durch eine einfache Transformation ein zugehöriges Meta-Programm $\mathcal{D}_{\mathcal{M}}(\Pi)$ generiert, das Rückschlüsse auf die Fehlerursachen liefert. Die Architektur des Meta-Programms wird schrittweise eingeführt, wobei begleitend wichtige Eigenschaften seiner Teilprogramme gezeigt werden. Abschließend wird auf die Formulierung verschiedener Debugging-Abfragen sowie auf den Einsatz von Optimierungstechniken eingegangen.

Eine weitere neue Technik zur Fehlersuche in Antwortmengenprogrammen wird in Kapitel 4 eingeführt. Sie beruht auf einer Transformation des fehlerhaften Programms, bei der tags, spezielle Kontroll- und Analyseatome, eingeführt werden. Neben der Beschreibung dieses Verfahrens wird auch auf die Unterschiede zum Ansatz von Kapitel 3 eingegangen.

Kapitel 5 behandelt das Software-Werkzeug `spock`, das sich als Prototyp zur Unterstützung der Fehlersuche in Antwortmengenprogrammen versteht. Neben allgemeinen Informationen zur Implementierung und deren Anwendung finden sich Beispiele zur Durchführung von Programm-Transformationen für die Fehlersuche.

Einen Überblick über kürzlich veröffentlichte Beiträge zu Debugging in der Antwortmengenprogrammierung liefert Kapitel 6. Es werden drei verschiedene Ansätze vorgestellt, diskutiert und gegenübergestellt.

Das letzte Kapitel dieser Arbeit bietet eine Zusammenfassung der behandelten Themen und gibt einen Ausblick auf mögliche künftige Forschungsansätze zum Thema Debugging in der Antwortmengenprogrammierung.

Teile dieser Arbeit wurden in Tagungsbänden internationaler Konferenzen und Workshops publiziert. Das Debugging-Verfahren in Kapitel 4 wurde auf der LPNMR'07 [5], der *9th In-*

ternational Conference on Logic Programming and Nonmonotonic Reasoning in Tempe, AZ, USA, präsentiert. Das Debugging-Tool `spock` wurde auf den Workshops SEA'07 [6], dem *1st Workshop for Software Engineering for Answer-Set Programming* in Tempe, AZ, USA, sowie dem WLP'07 [29], dem *21st Workshop on* (*Constraint*) *Logic Programming* in Würzburg, Deutschland, vorgestellt. Diese Arbeit wurde durch Mittel des *Fonds zur Förderung der wissenschaftlichen Forschung* (FWF) unter Projekt Nr. P18019 unterstützt.

Preface

This thesis deals with debugging in answer-set programming (ASP), a logic-programming paradigm which became popular in the scientific community for its fully declarative semantics. A major reason why ASP has not found widespread popularity as problem-solving technique yet is a lack of suitable tools for program development. In particular, debugging in ASP is an important field which has not been very well studied so far.

This work introduces two new techniques for finding errors in answer-set programs; one is based on *ASP-meta-programming*, the other relies on a *tagging technique*. Besides the description of the new approaches, an overview of existing publications concerning debugging of answer-set programs is given.

The method using meta-programming constitutes the main contribution of this work. It is based on a transformation of an erroneous program, Π, into the language of a meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$. The encoding allows for extracting exhaustive information about the original program Π. Each answer set of the meta-program describes the behaviour of Π under an interpretation I which is not an answer set of Π. Conversely, for each interpretation I of the original program not belonging to the answer sets of Π, there is a corresponding answer set of the meta-program. The key strategy of our approach is to point out reasons *why* I is not an answer set of Π. This information can be found in the answer sets of the meta-program, where we distinguish between four kinds of reason:

- the body of a rule is satisfied, although no element of the respective rule head is;
- an atom a is contained in I, although a is not in the head of any applicable rule;
- an integrity constraint is violated under I; or
- the truth of an atom in I is caused by itself.

The method allows to use answer-set programs in order to formulate complex debugging queries. In particular, the huge amount of information describing an erroneous situation can be reduced to parts relevant to the developer by using standard ASP-optimisation techniques.

The second new debugging method is adopted from a tagging technique for transforming ordered logic programs into standard ones. By introducing dedicated atoms, called *tags*, the applicability of individual rules can be controlled and several properties of the erroneous program can be read off the answer sets of a transformation of the program.

A significant feature of both new debugging strategies is their independence of the underlying algorithm for computing answer sets. The declarative view of answer-set programs can be maintained even when debugging, in contrast to other areas of logic programming, where errors are usually found by following the solving algorithm (cf. tracing in `PROLOG`).

We also developed a prototype of a debugging support tool for answer-set programs. The software, named `spock`, performs the program translations needed for the introduced debugging approaches.

The structure of this thesis is as follows:

Chapter 1 gives an introduction to the principles of ASP and the basics of error detection in programming. At first a short synopsis about the historical and general context of answer-set programming is given. After an overview about debugging in several programming paradigms, the peculiarities of debugging in ASP will be discussed.

The formal background of ASP is introduced in Chapter 2; besides syntax and semantics of answer-set programs, important properties of programs and further conventions are described, providing the theoretical basis for the following chapters.

The next chapter deals with a novel approach to debugging answer-set programs. From an erroneous program, Π, a simple transformation generates a corresponding meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$, from which we can draw conclusions about the origin of the occurring errors. The meta-program is introduced step-by-step, and important properties about its subprograms will be shown along with this process. Finally, we will discuss the formulation of various debugging queries and the usage of optimisation techniques.

Another new approach to finding errors in answer-set programs is presented in Chapter 4. It relies on a transformation of the erroneous program which introduces tags. These are special control and analysis atoms. Besides the description of the new method, the differences between the two new techniques will be discussed.

In Chapter 5, the software tool `spock` is presented, which is a prototype for supporting error detection in answer-set programs. In addition to general information about implementation and usage of `spock`, example applications of program translations for debugging are presented.

An overview of recently published contributions to debugging in ASP is given in the next chapter. Three different approaches will be presented, discussed and compared.

Chapter 7 closes this work, providing a summary of the topics addressed and delivering an outlook on further research into debugging in answer-set programming.

Parts of the work have been published in the proceedings of international conferences and workshops. The debugging method, discussed in Chapter 4, was presented at the *9th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'07) [5], in Tempe, AZ, USA. The debugging tool `spock` was demonstrated at the *1st Workshop for Software Engineering for Answer-Set Programming* (SEA'07) [6], in Tempe, AZ, USA, and at the *21st Workshop on (Constraint) Logic Programming* (WLP'07) [29], in Würzburg, Germany. This thesis was supported by the *Austrian Science Fund* (FWF) under project P18019.

Acknowledgements

I want to thank my supervisor Hans Tompits for his continuing support and advice, and for teaching me the art of writing scientific publications. I have learned a lot about style and presenting theoretical results. Furthermore, he spent a lot of time on our discussions about this work. Moreover, I am very grateful to him for giving me the possibility to visit the *9th International Conference on Logic Programming and Nonmonotonic Reasoning 2007* in Tempe, AZ, USA, the *Workshop on (Constraint) Logic Programming* in Würzburg, Germany, and the *Department of Knowledge Processing and Information Systems* at the University of Potsdam, Germany.

I highly appreciated the help of Stefan Woltran, who gave me lots of practical advice and hints for literature. In the early phase of writing my thesis, he was my primary source of advice and helped me to develop the meta-debugging framework which constitutes the main contribution of this work.

Furthermore, I would like to give special thanks to Veronika Peterseil for proofreading select parts of the work, Matthias Schlögel for his technical support, and to Elfriede Nemoda for her administrative work. I wish her all the best for her retirement.

The most important supporters have been my parents, Luise and Josef Pührer, who made me enjoy a good education and encouraged me to pursue my goals. Due to them, my life has always been free of existential worries, allowing me to concentrate on my studies (fulltime), which I clearly recognise as a privilege, unfortunately not granted to many others.

The *Austrian Science Fund* (FWF) covered my travel costs and student grants I have been given under project P18019, *Formal Methods for Optimizing Nonmonotonic Logic Programs*.

Contents

Chapter 1

Introduction

Logic programming emerged in the 1960s and 1970s, following the idea of representing knowledge in terms of logic and using mechanical deduction for problem solving. Based on Robinson's *resolution rule* [57] and work on automated theorem proving, Colmerauer and Kowalski developed the foundations of the programming language PROLOG (PROgramming in LOGic) in 1971 [38]. In the beginning, knowledge was expressed solely by *Horn clauses* which are disjunctions of literals, including mostly one positive literal. Rules of this kind allow fast and simple inferencing through resolution. Soon, the concept of the *closed-world assumption* in databases [54] was adopted to PROLOG, and the related *negation-as-failure* operator *not* was introduced to handle negative information. The idea is that a negated atom *not a* is considered to be true whenever a cannot be deduced. Due to this interpretation of negation, reasoning with PROLOG is nonmonotonic, e.g., adding a fact a to a knowledge-base falsifies *not a* and therefore a rule $b \leftarrow$ *not a* is not ensuring the truth of b any more.

A major novelty of the logic-programming paradigm was to have a declarative view on computer programs. Instead of stating how a problem should be solved, the problem is described by means of logical implications, leaving the imperative deduction process to the PROLOG interpreter. Kowalski expressed the relation of declarative and imperative aspects of a computation by the symbolic pseudo-formula $A(lgorithm) = L(ogic) + C(ontrol)$ [37]. In PROLOG the C-part is fixed, therefore the problem solving behaviour A is solely dependent on PROLOG programs, representing the L-part.

However, there are shortcomings in the declarativity of PROLOG, causing that programs can be seen as guiding instructions for the solving algorithm, rather than as pure problem descriptions. Essential deficits are the following:

- in PROLOG the order of literals in the body of a rule matters, as the interpreter processes them from left to right;
- in the same way, the position of a rule in a PROLOG program influences the obtained results;
- a program may cause infinite loops of the interpretation algorithm; and
- the PROLOG language includes the extra-logical *cut operator*.

From a theoretical point of view, the major difficulty in agreeing on a better semantics for logic programs is the handling of negation. Indeed, this is a non-trivial task, as the existence

of a unique least model, which is a key feature in Horn logic programming, is not guaranteed for programs involving negation.

Several semantics for logic programs have been proposed, aiming at overcoming the lack of declarativity in PROLOG and providing semantical clarity, also for negation. Many of these proposals were based on the desire to have a *single* intended model for every program. Apt, Blair, and Walker [3] introduced the *perfect-model semantics* which assigns a unique *perfect model* to logic programs with negation, satisfying the syntactic restriction of *stratification*. Another solution is given by the *well-founded semantics* by Van Gelder, Ross, and Schlipf [30]. Here, every program can be assigned a single intended model which is 3-valued, allowing the truth value of an atom to be *undefined*.

Other approaches abandon the demand for a canonical model and define collections of models as intended ones, e.g., *supported models*, introduced by Clark [12]. In 1988, Gelfond and Lifschitz [32] presented the *stable-model semantics*, also-called *answer-set semantics*, for normal logic programs following the ideas of default logic [55]. Three years later they extended it to disjunctive logic programs [33]. The intended models are referred to as *stable models* or *answer sets*. Generally, a program may have multiple or even no answer set. Based on this semantics, *answer-set programming* (ASP) emerged in the late 1990s as a new logic-programming paradigm [48, 46, 42], providing a fully declarative view on logic programs. In ASP, typically, a problem is described by a logic program such that the answer sets of the program correspond to the solutions of the problem. This is a major shift of perspective, as in classical logic programming, reasoning is performed by query answering rather than model generation. Nowadays, the answer-set semantics is the most popular declarative semantics for logic programs with nonmonotonic negation.

Answer sets are computed by *answer-set solvers*, such as DLV [23, 19, 40], SMODELS [48, 62], or CMODELS [41]. The latter two, like many other solvers, require prior external grounding of the input programs, which can be performed by the grounding frontend LPARSE [65, 66].

Answer-set programming has been successfully applied to many areas including planning [21, 16, 50], diagnosis [18, 31], product configuration [63], bounded model checking [35], agent systems [49, 9, 8, 4], and the Semantic Web [24, 60, 11].

1.1 Definition of Debugging

A definition of *debugging* is given in the ANSI/IEEE Standard Glossary of Software Engineering Terminology [2]:

> "Debugging is the process of locating, analyzing, and correcting suspected faults."

Furthermore, a *fault* is defined as

> "accidental condition that causes a program to fail its required function".

Thus, software debugging deals with finding and eliminating errors ("bugs") in computer programs. The nature of these errors is manifold and reaches from simple misspellings to conceptual programming errors. Software tools supporting debugging are referred to as *debuggers*.

According to a general belief, the first computer bug was a moth which caused a failure of the famous Mark-II computer in 1945. However, the term had been used even earlier for errors in technical devices [10].

1.2 Debugging in Logic Programming

The most prominent example for established debugging techniques in logic programming is *tracing* in `PROLOG`, enabling developers to control the evolution of the proof-search tree. Like traditional debugging techniques used in imperative programming, tracing relies on tracking the underlying steps in the execution of a program. The technique is based on the *"Byrd Box Model of Execution"* [13]. In this model, every goal predicate is represented by a box with four ports named `call`, `exit`, `redo`, and `fail`. A debugger reports `call` to the user when the attempt to recursively derivate a goal has started, and `exit` when the call has been successful, i.e., every subgoal has been derived. If a subgoal cannot be proved and the interpreter has to backtrack, `redo` will be reported. Finally, if every attempt to derive a predicate has not succeeded, the debugger outputs `fail`. The idea is that by following the control flow represented by these messages, the user recognises how the erroneous results emerge. As the search for a query might be very long, `breakpoints` or `spypoints` can be set, in order to skip large parts of the execution and start tracing exhaustively when a preselected line of code or a preselected predicate is to be evaluated.

1.3 Errors and Debugging in Answer-Set Programming

In order to illustrate the nature of errors in ASP, we will consider a classification scheme for errors, tailored to suit classical imperative programming languages. Herein, bugs are distinguished by the *level of specification* in which they occur in the program [1, 68]:

- *lexical* and *syntactic errors*: the program contains strings or sentences not occurring in the programming language;
- *semantic errors*: the program meets the syntactical requirements of the language but the assembly of its components does not make sense;
- *conceptual errors*: the program is correct but it does not serve the intended purpose.

Lexical and syntactic errors include misspellings of keywords, identifiers or operators, unbalanced parenthesis in arithmetic expressions. Examples for semantic errors are "division by zero", "infinite loops" and "index-out-of-bound errors". They are typically recognised in the first place when the program is executed. Therefore, these types of errors are also referred to as *runtime errors*. Conceptual errors are often first recognised when the program is systematically tested or already in application [59].

Now, we will view errors in ASP with regard to these categories of error. Brain and De Vos [7] pointed out that, due to the simple structure of ASP languages, the scope for lexical and syntactic errors is rather small. Notwithstanding, identifiers for predicates, variables, or terms can be mistaken or be spoiled by typos. This may sometimes cause hard trackable errors, as ASP languages usually do not enforce prior declaration of identifiers. Introducing a type system and respective checks would be straightforward and could help to avoid such bugs.

Furthermore, semantic errors do not exist in ASP, as every syntactically correct program has a well-defined semantics. Due to the declarativity of answer-set programs, we mainly deal with conceptual errors: mismatches between the actual and the intended semantics of a program. In terms of ASP, we encounter this phenomenon whenever the computed answer

sets do not match our expectations. In order to clarify what we mean by an error in ASP, we call a program having the intended semantics *correct*, whereas a program having a semantics not intended by the programmer is *incorrect*. At this abstract level, finding an error means to identify reasons for the incorrectness of a program in terms of properties of the program.

For more concrete characterisations of errors in ASP, we need to decide which general debugging question should be addressed. As we deal with discrepancies between an intended and an actual set of answer sets, one typical question is why an interpretation is an answer set, despite not being supposed to be one, or more general, why specific atoms are contained in an answer set, although they are not supposed to be. In such a scenario, errors can be, e.g., applicable rules deriving these atoms, other atoms making these rules applicable, or missing rules responsible for pruning the considered answer set. The related problem, why atoms are not contained in a specific answer set, is a special case of the question why a specific set of literals is not satisfied by any answer set of the incorrect program. Here, we can identify errors such as unsatisfied rules, unsupported atoms, violated integrity constraints, and positive or odd negative loops of dependency between atoms. Basically, different characterisations of the answer-set semantics allow different answers to the question why an interpretation is not an answer set.

So far, no established methodology for debugging answer-set programs has evolved. However, there have been some initial proposals concerning the issue [5, 7, 26, 67, 51]. A main distinguishing feature of possible debugging methods for ASP is whether the programmer needs to know the actual algorithm for computing answer sets for debugging. When applying techniques similar to tracing to ASP, such knowledge would be a basic requirement. This is problematic for several reasons. First of all, there is no canonical algorithm for answer-set computation. Some answer-set solvers implement a backtracking strategy [23, 48], typically equipped with advanced heuristics for faster computation; other systems transform the given program into another problem such as propositional satisfiability checking [41, 44] or solving of quantified boolean formulas [17]. Apart from the diverseness of computation strategies, the intermediate data obtainable during answer-set solving might be quite remote from the syntax and structure of the original program. This is especially the case in problem-transformation approaches. Finally, fixing a computation algorithm would impose an imperative view on answer-set programs, depriving the elegance of declarativity of the ASP paradigm.

1.4 Declarative Debugging

The concept of *declarative debugging* was originally introduced as *algorithmic debugging* by Shapiro [61] in 1982. The basic idea is that the debugging system detects errors guided by information about intended properties of the program. This information has to be supplied by an oracle, typically the programmer. Thus, the user has to supply declarative knowledge about the intended semantics of a program, but can prescind from the computational behaviour of the system.

Declarative debugging was initially used for debugging `PROLOG` programs, but has been proposed as a general approach towards debugging and also been applied to other paradigms, such as functional [47] and imperative programming [28]. Declarative debuggers usually traverse a tree representing a failed computation, called *proof tree* or *execution tree*, depending on the programming paradigm. When passing a vertice, the user is asked whether a specific part of the computation has been correct. If it was, the subtree below a correct vertice is

skipped in the traversal.

In Chapter 3 we will introduce a debugging technique for answer-set programs which follows the spirit of declarative debugging, as errors are detected by user knowledge of the expected results. However, in our approach, the programmer is not asked questions about the correctness of computed results, but specifies the intended results by means of ASP.

Chapter 2

Preliminaries

In this chapter we will introduce syntax, semantics, and important properties of answer-set programs. Although this thesis deals with debugging of propositional answer-set programs, we will introduce a language for the more general approach of *non-ground programming*. This is done because some of the discussed debugging techniques utilise non-ground programs. Furthermore, we will need special language constructs implemented in the answer-set solvers `DLV` and `SMODELS`.

2.1 Syntax

Logic programs are built from an alphabet which comprises identifiers for the objects we want to reason about and their properties of interest.

Definition 2.1 An *alphabet* for logic programs is a triple $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a non-empty set of *predicates symbols*, $\mathcal{V}$ is a set of *variables*, and $\mathcal{C}$ is a non-empty set, called the *domain*, whose elements are referred to as *constants*. It is assumed that each $p \in \mathcal{P}$ has an associated *arity* $\alpha(p) \geq 0$. The elements of the set $\mathcal{T} = \mathcal{C} \cup \mathcal{V}$ are called *terms*. □

By convention, variables are denoted by symbol strings starting with capital letters, constants by strings starting with numbers or lower case letters, and predicate symbols by strings starting with a letter. The arity of predicate symbol p can be indicated by appending a slash followed by the arity: $p/\alpha(p)$.

For convenience, we will sometimes use the so-called *anonymous-variable notation*, denoted by the underscore "_". Each occurrence of the anonymous variable stands for a new variable which is not used anywhere else in the considered context.

Example 2.2 Here are a few examples for predicate symbols, variables, and constants, respectively:

- *special*, *VerySpecial*, *strange2me*, *interesting_property*;
- X, *August15th*, *Bird*, *A_1*, *Why_not_using_long_variable_Names*;
- x, 0815, *tweety*, *constANT*, *this_Is_2_Nice*. ♢

Predicate symbols of arity n are used to express properties of tuples of n terms. A predicate symbol attached to such an n-tuple is an *atom*. In answer-set programming, the matter of reasoning is the question which combinations of atoms form answer sets of a program.

Definition 2.3 Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ be an alphabet for logic programs. An *atom* (*over* $\mathcal{A}$) is an expression of form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol with arity $\alpha(p) = n$ and $t_i \in \mathcal{T}$, for $1 \leq i \leq n$. We also say that $p(t_1, \ldots, t_n)$ is an atom *over* predicate p. A *literal* (*over* $\mathcal{A}$) is either an atom over $\mathcal{A}$ or an expression of form *not* a, where a is an atom over a and *not* is called *default negation*. Moreover, we say that a literal *not* a is a *negated atom*. □

An atom $p(t_1, \ldots, t_n)$ is *ground* iff each argument $t_i, 1 \leq i \leq n$, is a constant from $\mathcal{C}$, otherwise $p(t_1, \ldots, t_n)$ is *non-ground*. Furthermore, a literal *not* a is ground iff atom a is ground, non-ground otherwise.

We define the set of all ground atoms over a set of predicates with arguments from a set of constant as follows:

Definition 2.4 Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ be an alphabet for logic programs. Then, the set of all ground atoms over predicates $P \subseteq \mathcal{P}$ with arguments from $D \subseteq \mathcal{C}$ is given by

$$\mathcal{G}_{P,D} = \{p(c_1, \ldots, c_{\alpha(p)}) \mid p \in P \text{ and } c_1, \ldots, c_{\alpha(p)} \in D\}.$$

Moreover, the set of all ground atoms over $\mathcal{A}$ is given by $At(\mathcal{A}) = \mathcal{G}_{\mathcal{P},\mathcal{C}}$. □

Example 2.5 Assume an alphabet $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ with $\mathcal{C} = \mathbb{N}$, $X, Y \in \mathcal{V}$, and consider the sets $D = \{1, 2\}$ and $P = \{odd/1, equal/2\}$. Then,

- $odd(1), odd(9), equal(1, 2), equal(2, 2)$ are ground atoms,
- $odd(1), not\ odd(9), not\ equal(1, 2)$ are ground literals,
- $odd(X), equal(X, X), equal(X, Y), equal(5, X)$ are non-ground atoms,
- $odd(X), not\ equal(X, X), not\ equal(X, Y)$ are non-ground literals, and
- $\mathcal{G}_{P,D} = \{odd(1), odd(2), equal(1, 1), equal(1, 2), equal(2, 1), equal(2, 2)\}$ is the set of all ground atoms over predicates P with arguments from D. ♢

Definition 2.6 A (*disjunctive*) *rule, r,* (*over* $\mathcal{A}$) is a pair $\langle H, B \rangle$, where H is a set of atoms over $\mathcal{A}$ and B is a set of literals over $\mathcal{A}$ such that $H \cup B \neq \emptyset$. We will denote a rule of form $\langle \{h_1, \ldots, h_k\}, \{b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m\} \rangle$ by

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

□

For a rule r of form (1), we introduce the following notation:

- $head(r) = \{h_1, \ldots, h_k\}$ is the *head* of r,
- $body(r) = \{b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m\}$ is the *body* of r,
- $body^+(r) = \{b_1, \ldots, b_n\}$ is the *positive body* of r, and
- $body^-(r) = \{b_{n+1}, \ldots, b_m\}$ is the *negative body* of r.

Example 2.7 Consider an alphabet $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ with $\mathcal{C} = \mathbb{N}$, $\{odd/1, even/1, sum/3\} \subseteq \mathcal{P}$, $\{X, Y, Z\} \subseteq \mathcal{V}$, and the following rules over $\mathcal{A}$:

$$\begin{aligned} r_1 &= even(X) \leftarrow not\ odd(X)\,, \\ r_2 &= even(Z) \leftarrow odd(X), odd(Y), sum(X, Y, Z)\,. \end{aligned}$$

Then, rule r_1 says that a number which is not odd is even, and rule r_2 states that the sum of two odd numbers is an even number. ♢

In the following, we will introduce syntactic classification properties which rules may enjoy.

Definition 2.8 A rule of form (1) is called *non-disjunctive* iff $k \leq 1$, and *normal* iff $k = 1$. A rule r is *positive* iff $body^-(r) = \emptyset$, and a *fact* iff $body(r) = \emptyset$. Furthermore, r is an *integrity constraint* (or simply a *constraint*) iff $head(r) = \emptyset$. Finally, a rule r is *ground* iff every atom $a \in (head(r) \cup body^+(r) \cup body^-(r))$ is ground. □

In order to restrict the range of variables occurring in the head of a rule, we introduce the safety condition for rules. Considering only safe rules is reasonable, in order to avoid general statements about all objects of the domain, which might be troublesome when changing the domain.

Definition 2.9 A rule r is *safe* iff each variable occurring in $head(r) \cup body^-(r)$ also occurs in $body^+(r)$. □

Example 2.10 Assume an alphabet $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ with $\mathcal{C} = \mathbb{N}$, $\{odd/1, even/1, sum/3, lessOrEqual/2\} \subseteq \mathcal{P}$, $\{X, Y\} \subseteq \mathcal{V}$, and the rules:

$$\begin{aligned} r_1 &= lessOrEqual(1, X) \vee lessOrEqual(X, 1) \leftarrow\,, \\ r_2 &= \leftarrow odd(X), even(X)\,, \\ r_3 &= even(Y) \leftarrow sum(X, X, Y)\,. \end{aligned}$$

Then, r_1 is an unsafe fact, r_2 is a safe positive integrity constraint, and r_3 is a safe normal positive rule. ♢

Given the notion of safe rules, programs are defined in the following way:

Definition 2.11 A (*disjunctive*) *logic program over* $\mathcal{A}$ is a set of safe rules. □

The properties of being non-disjunctive, normal, positive, or ground can be extended straightforwardly from rules to programs.

Definition 2.12 A program is called *non-disjunctive* (respectively, *normal, positive, ground*) iff every rule in it is non-disjunctive (respectively, normal, positive, ground). □

While ground programs involve constants as arguments of atoms, propositional programs do not comprise any terms.

Definition 2.13 A program Π is *propositional* iff each predicate in Π has arity 0. □

By convention, we assume that, given an alphabet $\mathcal{A}$, every ground atom $a \in At(\mathcal{A})$ is viewed as a predicate symbol of arity 0. This way we can treat ground programs like propositional programs and vice versa.

Usually, the alphabet of an answer-set program Π is not given explicitly, but implicitly, determined by the constants and predicates occurring in Π. They determine the *Herbrand base* of Π which is the set of all atoms constructed from the predicates in Π, having constants occurring in Π as arguments.

Definition 2.14 Let Π be a program, P the set of predicate symbols occurring in Π, and C the set of constants occurring in Π. The *Herbrand universe*, $HU(\Pi)$, of Π is defined as follows:

$$HU(\Pi) = \begin{cases} C, & \text{if } C \neq \emptyset; \\ \{c\}, & \text{otherwise, with } c \in \mathcal{C} \text{ arbitrary.} \end{cases}$$

The *Herbrand base*, $HB(\Pi)$, of Π is the set of all ground atoms over predicates P with arguments from $HU(\Pi)$:

$$HB(\Pi) = \mathcal{G}_{P,HU(\Pi)}. \qquad \square$$

Since the answer-set semantics for non-ground programs will be defined in terms of the semantics for ground programs, we need the notation of the *grounding* of a program, transforming non-ground programs into ground ones.

By a *substitution* we understand a partial function $\theta : \mathcal{V} \to \mathcal{V} \cup \mathcal{C}$, where $\mathcal{V}$ is a set of variables and $\mathcal{C}$ is a set of constants. We say that a substitution θ is a *grounding* if, for all $v \in \mathcal{V}$, it holds that $\theta(v) \in \mathcal{C}$. Furthermore, given a rule r, by $r\theta$ we denote the rule resulting from r by replacing each $v \in \mathcal{V}$ in r by $\theta(v)$.

Definition 2.15 Let r be a rule and C a set of constants, then $Gr(r, C) = \{r\theta \mid \theta : V \to C\}$, where V is the set of variables occurring in r. Furthermore, for a program Π, we have $Gr(\Pi, C) = \bigcup_{r \in \Pi} Gr(r, C)$. Finally, the *grounding* (or *ground instantiation*), $Gr(\Pi)$, *of* Π is given by

$$Gr(\Pi) = Gr(\Pi, HU(\Pi)). \qquad \square$$

Observe that for every ground or propositional program Π, it holds that $Gr(\Pi) = \Pi$.

Example 2.16 Consider program Π for finding paths in a directed graph:

$$\begin{aligned} \Pi = \{\ & path(X, Y) \leftarrow edge(X, Y), \\ & path(X, Y) \leftarrow path(X, Z), edge(Z, Y), \\ & edge(a, b) \leftarrow, \\ & edge(b, a) \leftarrow, \\ & edge(b, c) \leftarrow\ \}. \end{aligned}$$

Then, the Herbrand universe $HU(\Pi)$ of Π consists of the three constants occurring in the atoms of Π, i.e., $HU(\Pi) = \{a, b, c\}$. Furthermore, the resulting grounding $Gr(\Pi)$ of Π con-

tains, e.g.:

$$
\begin{aligned}
&path(a,a) \leftarrow edge(a,a)\,,\\
&path(a,b) \leftarrow edge(a,b)\,,\\
&path(b,a) \leftarrow edge(b,a)\,,\\
&path(b,c) \leftarrow path(b,a),\, edge(a,c)\,,\\
&path(b,a) \leftarrow path(b,b),\, edge(b,a)\,,\\
&path(c,c) \leftarrow path(c,c),\, edge(c,c)\,,\\
&edge(a,b) \leftarrow\,,\\
&edge(b,a) \leftarrow\,,\\
&edge(b,c) \leftarrow\,.
\end{aligned}
$$

♢

2.2 Semantics

In this section, we describe the answer-sets semantics for disjunctive logic programs [33] which extends the stable-model semantics for normal logic programs [32].

Definition 2.17 Let $\mathcal{A}$ be an alphabet for logic programs. An *interpretation* is a set $I \subseteq At(\mathcal{A})$ of ground atoms. An interpretation I *satisfies* an atom a over $\mathcal{A}$ iff $a \in I$, and the negated atom *not* a iff $a \notin I$. Alternatively, we say that a literal is *true* under I if it is satisfied by I. Otherwise, it is *false* under I. We write $I \models l$ to express that literal l is true under I. Moreover, we say that I *satisfies* a set S of ground literals over $\mathcal{A}$ (symbolically $I \models S$) if every $l \in S$ is satisfied by I. Whenever $I \models body(r)$, for a ground rule r over $\mathcal{A}$, we say that r is *applicable under* I, and *blocked under* I otherwise. Furthermore, r is *satisfied by* I (symbolically $I \models r$) if r is blocked under I or an atom in $head(r)$ belongs to I, otherwise r is *unsatisfied by* I, or *violated under* I. I *satisfies* a ground program Π over $\mathcal{A}$ (symbolically $I \models \Pi$) if every rule in Π is satisfied by I. If $I \models \Pi$, we call I a *model* of Π. □

We will sometimes consider an arbitrary interpretation with respect to a given program. For this purpose we need the notion of an *interpretation for a program.*

Definition 2.18 Let Π be a disjunctive program and $I \subseteq HB(\Pi)$ a set of ground atoms. Then, I is an *interpretation for* Π. □

Identifying answer sets of a program Π includes a two-step transformation. The first step is computing the ground instantiation of Π. From that, we build the reduct of the program with respect to a candidate interpretation I which is a program not involving default negation.

Definition 2.19 The *reduct* of ground program Π with respect to a set I of ground atoms is the positive program Π^I, defined as follows:

$$\Pi^I = \{\ head(r) \leftarrow body^+(r)\ \mid r \in \Pi \text{ and } I \cap body^-(r) = \emptyset\}.$$ □

Definition 2.20 An interpretation I is an *answer set* of a program Π iff I is a minimal model of $Gr(\Pi)^I$.

The set of all answer sets of program Π is denoted by $AS(\Pi)$. Whenever $AS(\Pi) = \emptyset$, we say that Π is *inconsistent*, otherwise Π is *consistent*. Furthermore, by

$$\mathsf{GEN}(\Pi, I) = \{r \mid r \in \Pi \text{ and } I \models \mathit{body}(r)\},$$

we denote the *set of generating rules of* Π *under interpretation* I. □

Note that for every answer set A of program Π, it holds that $A \subseteq HB(\Pi)$.

For illustration, consider the following program:

$$\begin{aligned}\Pi_T = \{\ & \mathit{flies} \leftarrow \mathit{bird}, \mathit{not}\ \mathit{penguin}, \\ & \mathit{bird} \leftarrow \mathit{penguin}, \\ & \mathit{bird} \leftarrow \mathit{chicken}, \\ & \mathit{chicken} \vee \mathit{penguin} \leftarrow\ \}.\end{aligned}$$

This program has two answer sets, $\{\mathit{bird}, \mathit{chicken}, \mathit{flies}\}$ and $\{\mathit{bird}, \mathit{penguin}\}$.

There are also programs without any answer sets. Consider the following encoding Π_E of the Epimenides paradox:

$$\begin{aligned}\Pi_E = \{\ & \mathit{liesEpimenides} \leftarrow \mathit{allCretansLie}, \\ & \mathit{allCretansLie} \leftarrow \mathit{not}\ \mathit{liesEpimenides}\ \}.\end{aligned}$$

Clearly, no interpretation can satisfy both rules, hence $AS(\Pi_E) = \emptyset$.

In other cases, the empty set is itself an answer set:

$$\begin{aligned}\Pi_\emptyset = \{\ & b \leftarrow a, \\ & c \vee d \leftarrow b, \\ & e \leftarrow a, \mathit{not}\ b\ \}.\end{aligned}$$

For this program, we have $AS(\Pi_\emptyset) = \{\emptyset\}$. Note that whenever $\emptyset$ is an answer set, it is the only one, because of the minimality criterion for answer sets.

The following two propositions follow immediately from the definition of answer sets.

Proposition 2.21 *Let* Π *be a ground program and*

$$f \ = \ h_1 \vee \cdots \vee h_k \leftarrow$$

a fact in Π. *Then, for all* $A \in AS(\Pi)$, *it holds that* $A \cap \mathit{head}(f) \neq \emptyset$.

Informally, a constraint specifies a set of literals which cannot be satisfied simultaneously.

Proposition 2.22 *Let* Π *be a ground program and*

$$c \ = \ \leftarrow b_1, \ldots, b_n, \mathit{not}\ b_{n+1}, \ldots, \mathit{not}\ b_m,$$

a constraint in Π. *Then, for all* $A \in AS(\Pi)$, *it cannot hold that* c *is violated under* A.

Throughout this work, we will sometimes need the set of all integrity constraints in a program.

Definition 2.23 Given program Π, $\mathfrak{C}(\Pi)$ denotes the set of integrity constraints in Π:

$$\mathfrak{C}(\Pi) = \{c \mid c \in \Pi \text{ and } \mathit{head}(c) = \emptyset\}.$$ □

2.3 Alternative Characterisations

In 2004, Lin and Zhao introduced *loop formulas* which allow for computing the answer sets of a program by means of SAT solvers [44]. Based on their results, Lee developed a model-theoretic characterisation of answer sets [39]. We will exploit both approaches for debugging. Therefore, we will need the notion of the *dependency graph of a program* Π which is a graph expressing the interrelations between the atoms occurring in Π.

2.3.1 Dependency Graph and Loops

In the following we will introduce we will introduce basic concepts of graph theory as found in standard literature [34].

Definition 2.24 A *directed graph* G is an ordered pair $G = \langle V, E\rangle$ such that V is a set of *vertices* and $E \subseteq V \times V$ is a set of ordered pairs of vertices from V, called *edges*. We will assume that V and E are finite. A *subgraph* $G' = \langle V', E'\rangle$ *of* G is a directed graph such that $V' \subseteq V$ and $E' \subseteq E$.

A *directed labelled graph* G is an ordered triple $G = \langle V, E, L\rangle$ such that V is a set of *vertices*, L is a set of *labels*, and $E \subseteq V \times V \times L$ is a set of ordered triples $\langle v_1, v_2, l\rangle$, called labels, where v_1 and v_2 are vertices from V and l is a label from L. Again we will assume that V, E and L are finite. □

Definition 2.25 A *path* Ψ in a directed graph $G = \langle V, E\rangle$ from $v_1 \in V$ to $v_i \in V$ is a non-empty sequence $\Psi = v_1, v_2, \ldots, v_{i-1}, v_i$ of vertices $v_1, v_2, \ldots, v_{i-1}, v_i \in V$ such that $i \in \mathbb{N}$ and for $1 \leq j < i$, $e_j = \langle v_j, v_{j+1}\rangle \in E$. We call i the *length* of the path. □

Definition 2.26 We call a directed graph $G = \langle V, E\rangle$ *strongly connected* if, for every pair of vertices $\langle v_1, v_2\rangle$ in V, there is a path from v_1 to v_2 and a path from v_2 to v_1 in G. A *strongly connected component* of a directed graph is a maximal subgraph of the graph which is strongly connected. □

The following definitions and results are adopted from Lee's paper [39], except for the notion of *unsupported atoms* and Proposition 2.31.

Definition 2.27 The *dependency graph* of a ground program Π is a directed labelled graph $G = \langle V, E, L\rangle$, where $V = HB(\Pi)$ and $L = \{+, -\}$ such that, for all $a, b \in V$, $\langle a, b, +\rangle \in E$ iff there is a rule $r \in \Pi$ such that $a \in head(r)$ and $b \in body^+(r)$, and $\langle a, b, -\rangle \in E$ iff there is a rule $r \in \Pi$ such that $a \in head(r)$ and $b \in body^-(r)$. Furthermore, the *positive dependency graph* is the directed graph $G^+ = \langle V, E^+, L\rangle$, where $E^+ = \{\langle v_1, v_2\rangle \mid \langle v_1, v_2, +\rangle \in E\}$, and the *negative dependency graph* of Π is the directed graph $G^- = \langle V, E^-, L\rangle$, where $E^- = \{\langle v_1, v_2\rangle \mid \langle v_1, v_2, -\rangle \in E\}$. □

Based on this definition, we may identify the *loops* of a program. These are sets of atoms which mutually depend on each other.

Definition 2.28 Let Π be a ground program. A non-empty set A of atoms is called a *loop* of Π if, for every pair $\langle a_1, a_2\rangle$ of atoms in A, there exists a path from a_1 to a_2 in the positive dependency graph of Π such that all vertices in this path belong to A.

A loop Γ of Π is *trivial* if Γ consists of a single atom a such that the positive dependency graph of Π does not contain an edge from a to itself. A loop which is not trivial is called *non-trivial*. Furthermore, a program Π is *absolutely tight* iff every loop of Π is trivial. □

Note that, for every atom a that occurs in Π, the singleton set $\{a\}$ is a loop, according to Definition 2.28. The following proposition follows immediately from Definitions 2.26 and 2.28. It relates the notion of a loop to strongly connected components in the positive dependency graph:

Proposition 2.29 *Γ is a loop of ground program Π iff Γ is a non-empty subset of the set of all vertices within a strongly connected component of the positive dependency graph of Π.*

2.3.2 Support and External Support

We will also need the following concepts for Lee's characterisation of the answer-set semantics. We define the notion of support in the sense of Inoue and Sakama [36].

Definition 2.30 A set J of ground atoms is *supported* by ground program Π with respect to a set I of ground atoms if, for each atom $a \in J$, there is a rule r in Π such that $J \cap head(r) = \{a\}$ and $I \models body(r)$. A set I of ground atoms is *supported* by Π if I is supported by Π with respect to I. Given a set I of ground atoms, $a \in I$ is *unsupported* by Π with respect to I if there is no rule $r \in \Pi$ such that both, $I \cap head(r) = \{a\}$, and $I \models body(r)$. □

Note that, for every set I of ground atoms such that I is not supported by a ground program Π, there is some $a \in I$ such that a is unsupported by Π with respect to I.

Proposition 2.31 *Let Π be a ground non-disjunctive program and I a set of ground atoms. I is supported by Π if for every $a \in I$, $\{a\}$ is supported by Π with respect to I.*

Proof. Assume for every $a \in I$, $\{a\}$ is supported by Π with respect to I. Then, for each atom $a \in I$, there is a rule r in Π such that $\{a\} \cap head(r) = \{a\}$ and $I \models body(r)$. Since every rule in Π has at most one element in its head, we have for each atom $a \in I$ that there is a rule r in Π such that $head(r) = \{a\}$ and $I \models body(r)$, and thus $I \cap head(r) = \{a\}$ and $I \models body(r)$. Therefore, I is supported by Π. ■

Definition 2.32 A set J of ground atoms is *externally supported* by ground program Π with respect to a set I of atoms if there is a rule in Π such that $head(r) \cap J \neq \emptyset$, $I \models body(r)$, $body^+(r) \cap J = \emptyset$, and $I \cap (head(r) \setminus J) = \emptyset$. Otherwise, J is called *externally unsupported* by Π with respect to I. □

The following characterisation of answer sets constitutes the main theorem of Lee:

Theorem 2.33 ([39]) *For any ground disjunctive program Π and any set I of atoms, I is an answer set for Π iff I satisfies Π, and every loop Γ of Π, where $\Gamma \subseteq I$, is externally supported by Π with respect to I.*

For the class of absolutely tight programs, a further characterisation of answer sets is as follows, also due to Lee:

Theorem 2.34 ([39]) *For any absolutely tight ground program Π and any set I of atoms, I is an answer set for Π iff I satisfies Π and I is supported by Π.*

2.3.3 Completion and Loop Formulas

We will also need the related characterisation of answer sets by Lin and Zhao [44] for propositional normal logic programs which uses propositional logic. Answer sets are the models of the *completion* of the program in the sense of Clark [12] and the *loop formulas* of the program.

We need the following notions to map the body of a rule to a logic formula:

Definition 2.35 Let r be a rule in a propositional normal program such that $head(r) = \{a\}$. Then,

- $H_F(r) = a$ and
- $B_F(r) = \big(\bigwedge_{b \in body^+(r)} b\big) \wedge \big(\bigwedge_{c \in body^-(r)} \neg c\big)$. □

Definition 2.36 Let $\mathcal{A}$ be an alphabet for logic programs and Π a propositional normal logic program over $\mathcal{A}$. The completion of program Π is the set of propositional formulas $PF(\Pi) \cup CF(\Pi, \mathcal{A})$, where

$$PF(\Pi) = \big\{B_F(r) \rightarrow H_F(r) \mid r \in \Pi\big\} \quad \text{and}$$
$$CF(\Pi, \mathcal{A}) = \big\{a \rightarrow \bigvee_{r \in \Pi, head(r) = \{a\}} B_F(r) \mid a \in At(\mathcal{A})\big\}.$$

The set of *loop formulas* associated with a non-trivial loop Γ is

$$LF(\Pi, \Gamma) = \neg\big(\bigvee_{r \in R(\Pi,\Gamma)} B_F(r)\big) \rightarrow \bigwedge_{a \in \Gamma} \neg a \ ,$$

where $R(\Pi, \Gamma) = \{r \in \Pi \mid head(r) \subseteq \Gamma \text{ and } body^+(r) \cap \Gamma = \emptyset\}$. □

We denote the set of all non-trivial loops in Π by $loop_{\mathcal{NT}}(\Pi)$. The set of all loop formulas of Π is $LF(\Pi) = \{LF(\Pi, \Gamma) \mid \Gamma \in loop_{\mathcal{NT}}(\Pi)\}$. We can now formulate Lin-Zhao theorem as follows:

Theorem 2.37 ([44]) *Let $\mathcal{A}$ be an alphabet for logic programs and Π be a propositional normal logic program over $\mathcal{A}$. A set $I \subseteq At(\mathcal{A})$ of atoms is an answer set of Π iff I is a model of all formulas in $PF(\Pi) \cup CF(\Pi, \mathcal{A}) \cup LF(\Pi)$.*

2.4 Stratified Normal Programs

Another sort of programs, showing properties exploited in our work, is given by the class of *stratified programs*. Note that we will only consider normal programs in this context.

Definition 2.38 Let Π be a normal program. We call Π *stratified* iff there is an assignment $s(\cdot)$ of integers to the atoms in $HB(\Pi)$ such that, for each rule $r \in Gr(\Pi)$, the following holds: if a is the atom in the head of r and b (respectively, *not* b) occurs in r, then $s(a) \geq s(b)$ (respectively, $s(a) > s(b)$). □

Proposition 2.39 ([32]) *Let Π be a normal program such that Π is stratified. Then, Π has a unique answer set.*

2.5 Further Conventions

For simplicity, we will utilise a disequality predicate which is a built-in in many solvers. Intuitively, for variables X and Y, $X \neq Y$ is true exactly when X and Y are substituted by different constants. Furthermore, we use a comparison predicate, $\leq$ which is also supported by many solvers. For variables X and Y, $X \leq Y$ is true exactly when X and Y are substituted by constants a and b such that $\langle a, b\rangle$ is a member of an implicit strict total order on the considered domain.

Also for convenience, we do not distinguish between a set A of atoms and the set $\{a \leftarrow | \ a \in A\}$ of facts.

Moreover, we allow nested expressions of form *not not* a, where a is some atom, in the body of rules. Implicitly, *not not* a is replaced by *not* $a^\star$, where $a^\star$ is a globally new atom which is generally filtered out of computed answer sets, and an auxiliary rule $a^\star \leftarrow$ *not* a is added to the considered program. We also take advantage of (singular) *choice rules* [62] of form $\{a\} \leftarrow body(r)$ which are an abbreviation for $a \leftarrow body(r), not\ not\ a$.

Weak constraints [40], as implemented in `DLV`, are another special kind of rules, used in this thesis. They are of the form

$$:\sim body(r)\ [\,Weight : Level\,], \tag{2}$$

where *Weight* and *Level* are positive integers. The rule expresses that answer sets are preferred which do not satisfy $body(r)$. Let I be a set of atoms and r a weak constraint of form (2), then r is *violated* under I iff $I \models body(r)$. The values of *Weight* and *Level* are used to specify priorities such that weak constraints are considered more important than all weak constraints of a lower level and those of the same level, having a lower weight. Formally, this weighting mechanism can be described by an objective function $OB_\Pi(A)$ for ground program Π with weak constraints $WC(\Pi)$, and answer set A of $\Pi \setminus WC(\Pi)$ as follows, using an auxiliary function f_Π which maps leveled weights to weights without levels:

$$\begin{aligned} f_\Pi(1) &= 1, \\ f_\Pi(n) &= f_\Pi(n-1) * |\,WC(\Pi)| * w^\Pi_{max} + 1, \qquad n > 1, \\ OB_\Pi(A) &= \textstyle\sum_{i=1}^{l^\Pi_{max}} (f_\Pi(i) * \sum_{r \in N^\Pi_i(A)} weight(r)), \end{aligned}$$

where w^Π_{max} and l^Π_{max} denote the maximum weight and maximum level over the weak constraints in Π, respectively; $N^\Pi_i(A)$ denotes the set of weak constraints in level i that are violated under A, and $weight(r)$ denotes the weight of weak constraint r.

A set of atoms A is called an *optimal answer set of* Π if A is an answer set of $\Pi \setminus WC(\Pi)$ and $OB_\Pi(A)$ is minimal over all the answer sets of $\Pi \setminus WC(\Pi)$. The optimal answer sets of a non-ground program are given by the optimal answer sets of its grounding.

Chapter 3

Meta-Programming Approach

The following approach towards debugging of logic programs under the answer-set semantics is inspired by the idea of *algorithmic debugging*, using expected and actual outcomes of a program's evaluation for locating the source of emerging bugs. The debugging method can be applied to *propositional disjunctive programs* and is itself based on answer-set programming.

3.1 Basic Method

The central debugging question addressed here is, why interpretations of the considered program to debug, Π, are not answer sets of Π. We distinguish between four types of errors, for explaining why a particular interpretation I for Π, expected to be an answer set of Π, is not an answer set:

1. *Unsatisfied rules*: If a rule $r \in \Pi$ is unsatisfied by I, the logical implication represented by r is violated, and thus I is no classical model of Π.

2. *Not externally supported loops*: If a loop Γ of Π, where $\Gamma \subseteq I$, is not externally supported by Π with respect to I, the truth of the atoms in Γ is self-caused only such that the minimality criterion for answer sets is not met by I.

3. *Violated integrity constraints*: If the body of a constraint $c \in \Pi$ is satisfied by I, I cannot be an answer set of Π. Note that this is just a special case of a rule unsatisfied by I.

4. *Unsupported atoms*: If an atom $a \in I$ is not supported by Π with respect to I, there is no applicable rule in Π for deriving a exclusively. Note that errors of this kind are special cases of not externally supported loops, as $\{a\}$ is a trivial loop Γ of Π which is not externally supported by Π with respect to I.

This selection of error-types is redundant in the sense that for deciding whether an interpretation is an answer set, it suffices to detect unsatisfied rules and not externally supported loops of Π, as shown by Lee [39]. However, by allowing more types of errors, we get a more differentiated insight in the context of program Π under interpretation I. From a developer's point of view, integrity constraints play a rather different role than other rules, as they are used to restrict results, rather than to generate them. Therefore, we find it useful to handle their violations separately. Moreover, by this differentiation, our debugging technique allows

for restricting the search for errors to interpretations for Π which are not violating any integrity constraint. Also from the programmer's perspective, unsupported atoms may seem to be more palpable reasons for errors than not externally supported loops of Π. Furthermore, these two types of error are usually corrected in a different way, since correcting an externally unsupported loop involves the identification of multiple rules of Π involved in the loop, which is not necessary when coping with unsupported atoms.

In this approach, debugging is done by ASP-meta-interpretation. The considered propositional disjunctive program to debug, Π, is translated into a *non-disjunctive non-ground meta-program,* $\mathcal{D}_{\mathcal{M}}(\Pi)$. The debugging results can be read off the answer sets of $\mathcal{D}_{\mathcal{M}}(\Pi)$.

Each answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$ gives reasons why a particular interpretation I of Π is not an answer set of Π, and for every interpretation for Π which is not an answer set of Π, there is at least one such answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$, containing so-called *error-indicating atoms*. A rule $r \in \Pi$, unsatisfied by interpretation I, is detected by the occurrence of an atom $\mathit{unsatisfied}(l_r)$ in an answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$, associated to I. Furthermore, $\mathit{unsupported}(l_a)$ indicates an unsupported atom a, $\mathit{violated}(l_c)$ a violated integrity constraint c, and $\mathit{selfCaused}(l_a)$ an atom a in a not externally supported loop.

As stated, there can be more than one answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$ associated to a single interpretation I for Π. This is because the detection of not externally supported loops is done by considering one loop Γ of Π at most in every answer set of the meta-program and testing the atoms in Γ for external support. Note that for every interpretation for Π which is not an answer set of Π there is an answer set of the meta-program not considering any loop.

As an example, consider program Π_{ex}, consisting of the rules

$$\begin{aligned} r_1 &= \mathit{night} \vee \mathit{day} \leftarrow , \\ r_2 &= \mathit{bright} \leftarrow \mathit{candlelight} , \\ r_3 &= \leftarrow \mathit{night}, \mathit{bright}, \mathit{not}\ \mathit{torch\_on} , \\ r_4 &= \mathit{candlelight} \leftarrow . \end{aligned}$$

Π_{ex} has only the answer set$\{\mathit{candlelight}, \mathit{day}, \mathit{bright}\}$, however, the programmer expects also $I = \{\mathit{candlelight}, \mathit{night}, \mathit{bright}\}$ to be an answer set of Π_{ex}.

The answer sets of meta-program $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$, projected to the predicates $\mathit{int}/1$, $\mathit{violated}/1$, $\mathit{unsatisfied}/1$, $\mathit{unsupported}/1$, and $\mathit{selfCaused}/1$, include the following set:

$$S = \{\mathit{int}(l_{\mathit{bright}}), \mathit{int}(l_{\mathit{candlelight}}), \mathit{int}(l_{\mathit{night}}), \mathit{violated}(l_{r_3}), \mathit{unsatisfied}(l_{r_3})\}.$$

The atoms over the predicate $\mathit{int}/1$ describe the considered interpretation I. From the occurrence of $\mathit{violated}(l_{r_3})$ in S, we know that r_3 is an integrity constraint, violated under I. Since a violated constraint is a special case of an unsatisfied rule, we also have $\mathit{unsatisfied}(l_{r_3}) \in S$.

In this example, the question is why a specific interpretation I, which the programmer intends to be an answer set, is not an answer set of the considered program. We can restrict the answer sets of $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ to those which are associated to I, by joining $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ with constraints on the predicate $\mathit{int}/1$ specifying the considered interpretations:

$$\begin{aligned} &\leftarrow \mathit{not}\ \mathit{int}(\mathit{candlelight}), \quad \leftarrow \mathit{not}\ \mathit{int}(\mathit{night}), \quad \leftarrow \mathit{not}\ \mathit{int}(\mathit{bright}), \\ &\leftarrow \mathit{int}(\mathit{day}), \quad \leftarrow \mathit{int}(\mathit{torch\_on}). \end{aligned}$$

The addressed debugging problem can also be broadened from asking why a single interpretation is no answer set of program Π, to the question why a specified class of interpretations for Π do not contain answer sets of Π. This class can be defined by programs using

the meta-atoms of $\mathcal{D}_{\mathcal{M}}(\Pi)$. Here, we may combine various criteria for choosing between the considered interpretations, e.g. we can select all interpretations in which specific atoms are (not) contained, specific rules are (not) applicable, and specific errors occur. Note, however, that the resulting answer sets of the meta-program refer to the specified interpretations for Π individually, not to the class as a whole. Subsection 3.4 will deal with similar and more advanced restrictions of the search-space.

Generally, ASP-meta-programming is a powerful approach to analysing programs which has been used previously in various contexts [25, 20, 45]. The meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ can easily extended to address several sorts of debugging requests, e.g., subprograms of $\mathcal{D}_{\mathcal{M}}(\Pi)$ can also be used to investigate the context of actual answer sets of Π.

3.2 Overall Architecture

Before presenting the technical details of this approach, it is helpful to provide a short overview of the architecture of the meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ for Π.

The program $\mathcal{D}_{\mathcal{M}}(\Pi)$ can be partitioned into several modules π_χ, where

$$\chi \in \{in(\Pi), aux, int, ap, dpcy, loop, ic, supp, sat, ext, noAS\},$$

each of which serving a different purpose. The way they are designed allows us to introduce the overall program step by step, using the properties of *potential usage* [22] between the modules.

In the following, we will shortly describe the intuitive meanings of the individual modules π_χ of $\mathcal{D}_{\mathcal{M}}(\Pi)$:

- $\pi_{in(\Pi)}$ provides an encoding of the original program Π to debug. It consists of facts, stating for each rule $r \in \Pi$ which atoms in $\mathcal{A}$ occur in the head, the positive body, and the negative body of r, respectively. $\pi_{in(\Pi)}$ can be seen as the input part of $\mathcal{D}_{\mathcal{M}}(\Pi)$ and is used by all other modules for reasoning about Π.
- π_{aux} is a set of auxiliary meta-rules, identifying all rules and atoms of Π.
- π_{int} guesses an interpretation I for Π. Thus, other modules may identify properties of Π under I.
- π_{ap} checks for each rule of Π whether it is applicable or blocked under interpretation I which was guessed by module π_{int}.
- π_{dpcy} derives the positive dependency graph of Π and detects its strongly connected components.
- π_{loop} guesses a loop Γ within a strongly connected component of the positive dependency graph of Π.
- π_{ic} detects integrity constraints of Π which are violated under interpretation I.
- π_{supp} detects atoms in interpretation I which are unsupported by Π with respect to I.
- π_{sat} detects rules of Π for which the respective bodies are not satisfied by interpretation I.

- π_{ext} checks whether loop Γ, guessed by module π_{loop}, is externally supported by Π with respect to interpretation I.

- π_{noAS} is used to filter out all meta-answer-sets where no error was detected.

The individual modules can be joined to subprograms $\mathcal{D}_\chi(\Pi)$ of $\mathcal{D}_\mathcal{M}(\Pi)$, $\chi \in \{in(\Pi), aux, int, ap, dpcy, loop, ic, supp, sat, ext, noAS\}$, according to an auxiliary relation $\succ_\mathcal{D}$, specifying which modules require which other modules to achieve meaningful results. E.g., guessing an interpretation I for Π is needed for detecting violated constraints of Π under I, thus module π_{int} is a prerequisite of module π_{ic}, i.e., $\pi_{int} \succ_\mathcal{D} \pi_{ic}$ holds.

The intuitive meaning of a subprogram $\mathcal{D}_\chi(\Pi)$ of $\mathcal{D}_\mathcal{M}(\Pi)$ is the same as described above for π_χ. However, $\mathcal{D}_\chi(\Pi)$ can be used as stand-alone program, while the individual modules are just program fragments which require further input in order to provide useful debugging information.

We will introduce the notion of *meta-answer-sets for* Π which refers to specific supersets of the answer sets of subprograms $\mathcal{D}_\chi(\Pi)$. According to which modules are a subset of $\mathcal{D}_\chi(\Pi)$, meta-answer-sets provide different features, like, e.g., checking of rule applicability, detection of loops, or detection of unsatisfied rules. The answer sets of the overall program $\mathcal{D}_\mathcal{M}(\Pi)$ incorporate all these features.

In the course of introducing $\mathcal{D}_\mathcal{M}(\Pi)$ step by step, we will prove various properties of the respective subprograms which are built on each other.

3.3 Meta-Program

3.3.1 Meta-Programs and Meta-Answer-Sets

For reasoning about a program Π over $\mathcal{A}$ within another answer-set program, an adequate alphabet is needed for expressing Π on the meta-level. The following definition introduces such an alphabet, $\mathcal{A}_\mathcal{M}(\mathcal{A})$, depending on the original alphabet $\mathcal{A}$ for Π, and specifies the class of programs which we use for debugging.

Definition 3.1 Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ be an alphabet for propositional disjunctive logic programs and $\mathcal{L}$ the set of all rules over $\mathcal{A}$. Then, a *meta-program* (*over* $\mathcal{A}$) is a non-ground non-disjunctive program over $\mathcal{A}_\mathcal{M}(\mathcal{A})$, where $\mathcal{A}_\mathcal{M}(\mathcal{A}) = \langle \mathcal{P}_\mathcal{M}, \mathcal{V}_\mathcal{M}, \mathcal{C}_\mathcal{M} \rangle$,

$$\begin{aligned}\mathcal{P}_\mathcal{M} = \{ & head/2, bodyP/2, bodyN/2, atom/1, rule/1, int/1, \overline{int}/1, ap/1, bl/1, dpcy/2, \\ & strCon/2, loop/1, \overline{loop}/1, violated/1, hasHead/1, otherHeadInI/2, oHOfApRinI/1, \\ & unsupported/1, anyHeadInI/1, unsatisfied/1, extSupp/0, anyBInLoop/2, \\ & hInINotLoop/1, selfCaused/1, noAnswerSet/0\},\end{aligned}$$

$\mathcal{V}_\mathcal{M}$ is a set of variables, and $\mathcal{C}_\mathcal{M}$ is a set of constants such that there is a bijection $l : \mathcal{L} \cup \mathcal{C} \rightarrow \mathcal{C}_\mathcal{M}$. We will denote the value $l(x)$ also by l_x. □

Intuitively the elements of $\mathcal{C}_\mathcal{M}$ represent labels for the rules and atoms over $\mathcal{A}$. Therefore, the atoms and rules of programs over $\mathcal{A}$ are the objects of reasoning in meta-programs over $\mathcal{A}$. The informal meanings of the predicate symbols in $\mathcal{P}_\mathcal{M}$ is given in Table 3.1.

We will call an answer set of a meta-program a *meta-answer-set*:

Predicate	Argument(s)	Intended meaning
$head/2$	l_r, l_a	Atom a is in the head of rule r.
$bodyP/2$	l_r, l_a	Atom a is in the positive body of rule r.
$bodyN/2$	l_r, l_a	Atom a is in the negative body of rule r.
$atom/1$	l_a	a is an atom in program Π to debug.
$rule/1$	l_r	r is a rule in Π.
$int/1$	l_a	a is element of considered interpretation I.
$\overline{int}/1$	l_a	a is not element of considered interpretation I.
$ap/1$	l_r	Rule r is applicable under I.
$bl/1$	l_r	Rule r is blocked under I.
$dpcy/2$	l_{a_1}, l_{a_2}	Atom a_1 positively depends on atom a_2.
$strCon/2$	l_{a_1}, l_{a_2}	Atoms a_1, a_2 are members of the same strongly connected component within the positive dependency graph of Π.
$loop/1$	l_a	Atom a is a member of the considered loop Γ of Π.
$\overline{loop}/1$	l_a	Atom a is not a member of the considered loop Γ of Π.
$violated/1$	l_r	Rule r is a violated integrity constraint.
$hasHead/1$	l_r	The head of rule r is not empty.
$otherHeadInI/2$	l_r, l_a	There is some b, $b \neq a$ such that $a, b \in head(r) \cap I$ for rule r.
$oHOfApRinI/1$	l_a	There is an applicable rule r for which a is the only atom such that $a \in head(r) \cap I$.
$unsupported/1$	l_a	Atom a is unsupported by Π with respect to the considered interpretation I.
$anyHeadInI/1$	l_r	There is an atom which is in the head of rule r and in I.
$unsatisfied/1$	l_r	The body of rule r is satisfied but there is no atom which is in the head of r and in I.
$extSupp/0$		Considered loop Γ is externally supported.
$anyBInLoop/2$	l_r, l_a	There is an atom in the body of rule r which is in the same loop as atom a.
$hInINotLoop/1$	l_r	There is some atom a which is not in the considered loop such that $a \in head(r) \cap I$.
$selfCaused/1$	l_r, l_a	Atom a is in the considered interpretation I but in a loop which is not externally supported.
$noAnswerSet/0$		Considered interpretation I is no answer set of Π.

Table 3.1: Predicates of $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$ and their meanings

Definition 3.2 Let $\mathcal{A}$ be an alphabet for propositional disjunctive logic programs, and $\Pi_{\mathcal{M}}$ a meta-program over $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$. We call every set $A_{\mathcal{M}} \in AS(\Pi_{\mathcal{M}})$ a *meta-answer-set* (*over* $\mathcal{A}$). Furthermore, we will refer to atoms over $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$ as *meta-atoms* (*over* $\mathcal{A}$). □

In what follows, we will characterise specialised classes of meta-answer-sets, providing information about a program Π over $\mathcal{A}$ to debug. To this end, we introduce the notion of a Meta-Program for Π in the next subsection.

3.3.2 Meta-Program for Π

Definition 3.3 Given a propositional disjunctive logic program Π over alphabet $\mathcal{A}$, the *meta-program for* Π is a meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ over $\mathcal{A}$ for Π such that

$$\mathcal{D}_{\mathcal{M}}(\Pi) = \pi_{in(\Pi)} \cup \pi_{aux} \cup \pi_{int} \cup \pi_{dpcy} \cup \pi_{ap} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext} \cup \pi_{noAS},$$

where each π_χ denotes a submodule of $\mathcal{D}_{\mathcal{M}}(\Pi)$ which will be defined and discussed in the subsequent subsections. □

We will sometimes use the set $\mathcal{MOD}(\Pi)$ of module names which is given by

$$\mathcal{MOD}(\Pi) = \{in(\Pi), aux, int, ap, dpcy, loop, ic, supp, sat, ext, noAS\}.$$

The complete program can be found in Figure 3.1. For convenience, we call the meta-program for Π, $\mathcal{D}_{\mathcal{M}}(\Pi)$, simply *the* meta-program, whenever Π is known from the context or irrelevant.

3.3.3 Potential-Use Relation and Module Prerequisites

The meta-program is composed in a modular fashion which allows us to introduce it step by step, by consecutively joining its modules π_χ. Therefore, we make use of the notion of *potential usage* of programs [22] which is closely related to splitting sets [43].

Definition 3.4 Let Π_1 and Π_2 be disjunctive programs. We say that Π_2 *potentially uses* Π_1 (denoted $\Pi_2 \rhd \Pi_1$) iff each atom that occurs in some head of Π_2 does not occur in Π_1. □

Later on, we will need the following proposition:

Proposition 3.5 *Let* Π_1, Π_2 *and* Π_3 *be disjunctive programs. Then,*

1. $(\Pi_2 \cup \Pi_3) \rhd \Pi_1$ *iff* $\Pi_2 \rhd \Pi_1$ *and* $\Pi_3 \rhd \Pi_1$, *and*
2. $\Pi_3 \rhd (\Pi_1 \cup \Pi_2)$ *iff* $\Pi_3 \rhd \Pi_1$ *and* $\Pi_3 \rhd \Pi_2$.

Proof. Part 1: ($\Rightarrow$) Assume that $(\Pi_2 \cup \Pi_3) \rhd \Pi_1$ holds. Then, by Definition 3.4, no atom in the head of any rule in $\Pi_2 \cup \Pi_3$ occurs in Π_1. Therefore, no atom in the head of any rule in Π_2 occurs in Π_1 and no atom in the head of any rule in Π_3 occurs in Π_1. Thus, again by Definition 3.4, it holds that $\Pi_2 \rhd \Pi_1$ and $\Pi_3 \rhd \Pi_1$.

($\Leftarrow$) Assume that $\Pi_2 \rhd \Pi_1$ and $\Pi_3 \rhd \Pi_1$ holds. Then, by Definition 3.4, no atom in the head of any rule in Π_2 occurs in Π_1 and no atom in the head of any rule in Π_3 occurs in Π_1. Therefore, no atom in the head of any rule in $\Pi_2 \cup \Pi_3$ occurs in Π_1. Thus, again by Definition 3.4, it holds that $(\Pi_2 \cup \Pi_3) \rhd \Pi_1$.

$\{head(l_r, l_a) \leftarrow\!| \; a \in head(r) \text{ and } r \in \Pi\} \cup$
$\{bodyP(l_r, l_a) \leftarrow\!| \; a \in body^+(r) \text{ and } r \in \Pi\} \cup$
$\{bodyN(l_r, l_a) \leftarrow\!| \; a \in body^-(r) \text{ and } r \in \Pi\} \cup$
$\{atom(A) \leftarrow head(\_, A)\,,$
$\; atom(A) \leftarrow bodyP(\_, A)\,,$
$\; atom(A) \leftarrow bodyN(\_, A)\,,$
$\; rule(R) \leftarrow head(R, \_)\,,$
$\; rule(R) \leftarrow bodyP(R, \_)\,,$
$\; rule(R) \leftarrow bodyN(R, \_)\,,$
$\; int(A) \leftarrow atom(A), not\ \overline{int}(A)\,,$
$\; \overline{int}(A) \leftarrow atom(A), not\ int(A)\,,$
$\; ap(R) \leftarrow not\ bl(R), rule(R)\,,$
$\; bl(R) \leftarrow bodyN(R, A), int(A)\,,$
$\; bl(R) \leftarrow bodyP(R, A), not\ int(A)\,,$
$\; dpcy(A, A) \leftarrow atom(A)\,,$
$\; dpcy(A1, A2) \leftarrow head(R, A1), bodyP(R, A2)\,,$
$\; dpcy(A1, A2) \leftarrow dpcy(A1, A3), dpcy(A3, A2)\,,$
$\; strCon(A1, A2) \leftarrow dpcy(A1, A2), dpcy(A2, A1)\,,$
$\; loop(A) \leftarrow not\ \overline{loop}(A), atom(A)\,,$
$\; \overline{loop}(A) \leftarrow not\ loop(A), atom(A)\,,$
$\; \leftarrow loop(A1), loop(A2), not\ strCon(A1, A2)\,,$
$\; violated(R) \leftarrow ap(R), not\ hasHead(R)\,,$
$\; hasHead(R) \leftarrow head(R, \_)\,,$
$\; otherHeadInI(R, A) \leftarrow head(R, A), head(R, A2), int(A), int(A2), A \neq A2\,,$
$\; oHOfApRinI(A) \leftarrow ap(R), int(A), head(R, A), not\ otherHeadInI(R, A)\,,$
$\; unsupported(A) \leftarrow int(A), not\ oHOfApRinI(A)\,,$
$\; anyHeadInI(R) \leftarrow head(R, A), int(A)\,,$
$\; unsatisfied(R) \leftarrow ap(R), not\ anyHeadInI(R)\,,$
$\; extSupp \leftarrow head(R, A), ap(R), loop(A), not\ anyBInLoop(R), not\ hInINotLoop(R)\,,$
$\; anyBInLoop(R) \leftarrow bodyP(R, A), loop(A)\,,$
$\; hInINotLoop(R) \leftarrow head(R, A), int(A), not\ loop(A)\,,$
$\; loopNotInI \leftarrow loop(A), not\ int(A)\,,$
$\; selfCaused(A) \leftarrow loop(A), not\ loopNotInI, not\ extSupp\,,$
$\; noAnswerSet \leftarrow unsatisfied(\_)\,,$
$\; noAnswerSet \leftarrow selfCaused(\_)\,,$
$\; \leftarrow not\ noAnswerSet\}$

Figure 3.1: Meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$

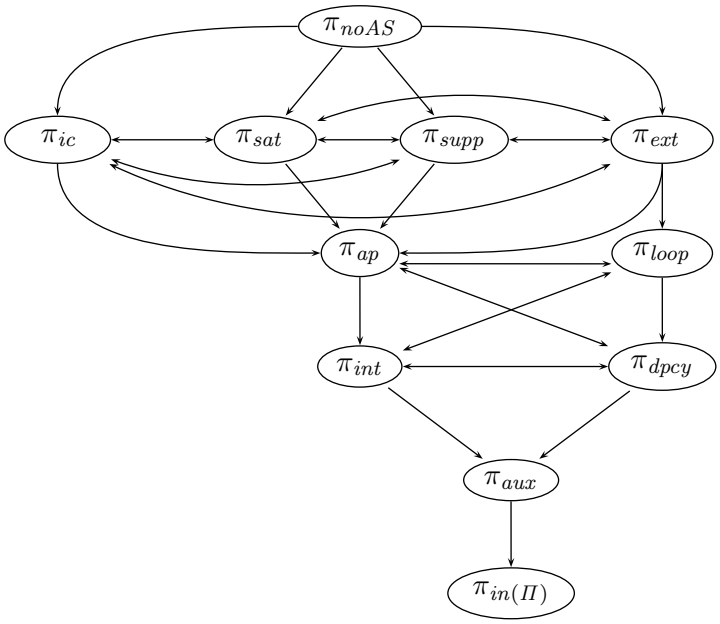

Figure 3.2: Partial potential use relation in $\mathcal{D}_{\mathcal{M}}(\Pi)$

Part 2: ($\Rightarrow$) Assume that $\Pi_3 \rhd (\Pi_1 \cup \Pi_2)$ holds. Then, by Definition 3.4, no atom in the head of any rule in Π_3 occurs in $\Pi_1 \cup \Pi_2$. Therefore, no atom in the head of any rule in Π_3 occurs in either Π_1 nor in Π_2. Thus, again by Definition 3.4, it holds that $\Pi_3 \rhd \Pi_1$ and $\Pi_3 \rhd \Pi_2$.

($\Leftarrow$) Assume that $\Pi_3 \rhd \Pi_1$ and $\Pi_3 \rhd \Pi_2$ hold. Then, by Definition 3.4, no atom in the head of any rule in Π_3 occurs in either Π_1 nor in Π_2. Therefore, no atom in the head of any rule in Π_3 occurs in $\Pi_1 \cup \Pi_2$. Thus, again by Definition 3.4, it holds that $\Pi_3 \rhd (\Pi_1 \cup \Pi_2)$. ■

Whenever a program Π_2 potentially uses another program Π_1, the answer sets of the joined program, $\Pi_1 \cup \Pi_2$, can be computed from Π_2 and the answer sets of Π_1 in the following way:

Proposition 3.6 ([22, 43]) *Let $\Pi = \Pi_1 \cup \Pi_2$ be a disjunctive program such that $\Pi_2 \rhd \Pi_1$. Then, it holds that*

$$AS(\Pi) = \bigcup_{A \in AS(\Pi_1)} (AS(A \cup \Pi_2)).$$

We will exploit Proposition 3.6 and potential usage between modules of $\mathcal{D}_{\mathcal{M}}(\Pi)$ to gradually built up $\mathcal{D}_{\mathcal{M}}(\Pi)$. Figure 3.2 illustrates the potential use relation between the modules (an arrow from π_χ to π_φ indicates $\pi_\chi \rhd \pi_\varphi$). Note that only edges of interest are plotted here.

Informally speaking, each module π_χ (except for $\pi_{in(\Pi)}$) requires the output of other modules in order to give reasonable results, as outlined in Section 3.2. These dependencies are captured by the transitive *prerequisites relation*, $\succ_{\mathcal{D}}$, which is a subrelation of $\rhd$.

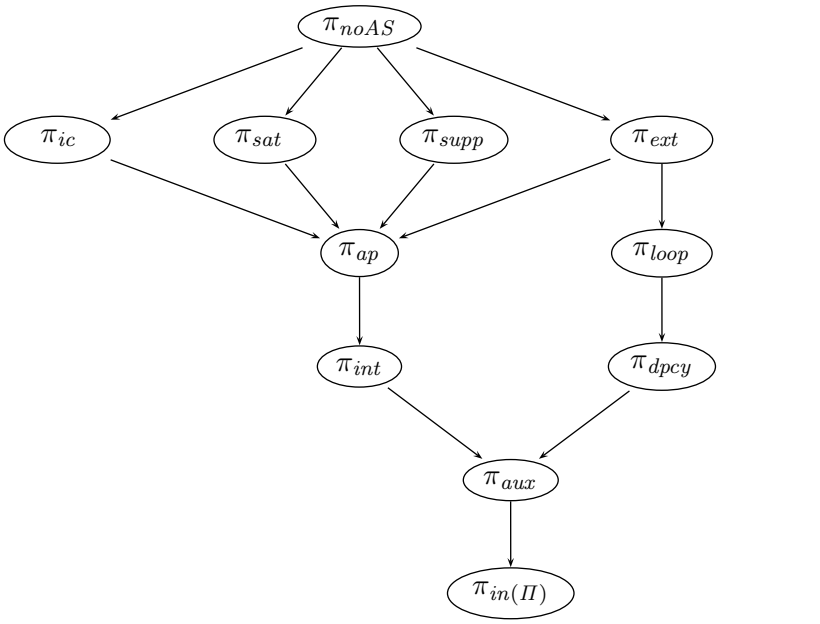

Figure 3.3: Prerequisite relation $\succ_{\mathcal{D}}$ between modules in $\mathcal{D}_{\mathcal{M}}(\Pi)$

Definition 3.7 The *prerequisite relation*, $\succ_{\mathcal{D}}$, is given by

$$\begin{aligned}\mathcal{B} = \{\ & (\pi_{aux}, \pi_{in(\Pi)}), (\pi_{int}, \pi_{aux}), (\pi_{dpcy}, \pi_{aux}), (\pi_{ap}, \pi_{int}), (\pi_{loop}, \pi_{dpcy}),\\ & (\pi_{ic}, \pi_{ap}), (\pi_{supp}, \pi_{ap}), (\pi_{sat}, \pi_{ap}), (\pi_{ext}, \pi_{ap}), (\pi_{ext}, \pi_{loop}),\\ & (\pi_{noAS}, \pi_{ic}), (\pi_{noAS}, \pi_{supp}), (\pi_{noAS}, \pi_{sat}), (\pi_{noAS}, \pi_{ext})\},\\ \succ_{\mathcal{D}} = {} & \mathcal{B}^{+},\end{aligned}$$

where $\mathcal{B}^{+}$ is the transitive closure of $\mathcal{B}$. If $\pi_{\chi} \succ_{\mathcal{D}} \pi_{\varphi}$ holds, for $\chi, \varphi \in \mathcal{MOD}(\Pi)$, then π_{φ} is a *prerequisite* for π_{χ}. □

A better overview of this relation is provided in Figure 3.3 (again, an arrow from π_{χ} to π_{φ} indicates $\pi_{\chi} \succ_{\mathcal{D}} \pi_{\varphi}$). Intuitively, $\pi_{\chi} \succ_{\mathcal{D}} \pi_{\varphi}$ expresses that the atoms derived by module π_{φ} are semantically needed as input for module π_{χ}.

3.3.4 Standalone Subprograms and Specialised Meta-Answer-Sets

Given the formalisation of module prerequisites, we can define subprograms $\mathcal{D}_{\chi}(\Pi)$ of $\mathcal{D}_{\mathcal{M}}(\Pi)$, consisting of modules such that all dependencies are fulfilled.

Definition 3.8 $\mathcal{D}_{\chi}(\Pi)$ is the union of module π_{χ} and all of its prerequisites:

$$\mathcal{D}_{\chi}(\Pi) = \pi_{\chi} \cup \bigcup_{\pi_{\chi} \succ_{\mathcal{D}} \pi_{\varphi}} \pi_{\varphi},$$

where $\chi \in \mathcal{MOD}(\Pi)$. □

For example:

$$\begin{aligned}\mathcal{D}_{sat}(\Pi) &= \pi_{sat} \cup \pi_{ap} \cup \pi_{int} \cup \pi_{aux} \cup \pi_{in(\Pi)},\\ \mathcal{D}_{loop}(\Pi) &= \pi_{loop} \cup \pi_{dpcy} \cup \pi_{aux} \cup \pi_{in(\Pi)}, \quad \text{and}\\ \mathcal{D}_{noAS}(\Pi) &= \mathcal{D}_{\mathcal{M}}(\Pi).\end{aligned}$$

Based on the programs $\mathcal{D}_\chi(\Pi)$, we define corresponding categories of meta-answer-sets:

Definition 3.9 Let Π be a propositional disjunctive program over alphabet $\mathcal{A}$. We say that a set $A_\mathcal{M}$ of atoms over $\mathcal{A}_\mathcal{M}(\mathcal{A})$ is a *meta-answer-set for* Π *with respect to* χ, or a *χ-MAS* for Π, iff $A_\mathcal{M} \in AS(\mathcal{D}_\chi(\Pi) \cup \mathcal{D}')$, where $\mathcal{D}'$ is a program over $\mathcal{A}_\mathcal{M}(\mathcal{A})$ such that $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$ and $\chi \in \mathcal{MOD}(\Pi)$.

A set $A_\mathcal{M}$ of atoms over $\mathcal{A}_\mathcal{M}(\mathcal{A})$ is called a *meta-answer-set for* Π iff $A_\mathcal{M}$ is a χ-MAS for Π, for some $\chi \in \mathcal{MOD}(\Pi)$. □

Proposition 3.10 *Let Π be a propositional disjunctive program over alphabet $\mathcal{A}$, $A_\mathcal{M}$ a set of atoms over $\mathcal{A}_\mathcal{M}(\mathcal{A})$, and $\chi \in \mathcal{MOD}(\Pi)$. If $A_\mathcal{M}$ is a χ-MAS for Π, then $A_\mathcal{M}$ is also a φ-MAS for Π, for all $\varphi \in \mathcal{MOD}(\Pi)$ such that $\pi_\chi \succ_\mathcal{D} \pi_\varphi$.*

Proof. Consider some $\chi, \varphi \in \mathcal{MOD}(\Pi)$ such that $\pi_\chi \succ_\mathcal{D} \pi_\varphi$, and assume that $A_\mathcal{M}$ is a χ-MAS for Π. By Definition 3.9, it holds that

$$A_\mathcal{M} \in AS(\mathcal{D}_\chi(\Pi) \cup \mathcal{D}'),$$

where $\mathcal{D}'$ is a program over $\mathcal{A}_\mathcal{M}(\mathcal{A})$ such that $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$. By Definition 3.8, we have that

$$\mathcal{D}_\chi(\Pi) = \pi_\chi \cup \bigcup_{\pi_\chi \succ_\mathcal{D} \pi_\varsigma} \pi_\varsigma.$$

Since $\pi_\chi \succ_\mathcal{D} \pi_\varphi$, and by the transitivity of $\succ_\mathcal{D}$, we can rewrite the last statement as follows:

$$\mathcal{D}_\chi(\Pi) = \pi_\chi \cup \Big(\bigcup_{\pi_\chi \succ_\mathcal{D} \pi_\varsigma \succ_\mathcal{D} \pi_\varphi} \pi_\varsigma\Big) \cup \pi_\varphi \cup \bigcup_{\pi_\varphi \succ_\mathcal{D} \pi_\zeta} \pi_\zeta.$$

Again, by Definition 3.8, we get

$$\mathcal{D}_\chi(\Pi) = \pi_\chi \cup \Big(\bigcup_{\pi_\chi \succ_\mathcal{D} \pi_\varsigma \succ_\mathcal{D} \pi_\varphi} \pi_\varsigma\Big) \cup \mathcal{D}_\varphi(\Pi).$$

Therefore, we can rewrite $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$ as

$$\mathcal{D}' \rhd \pi_\chi \cup \Big(\bigcup_{\pi_\chi \succ_\mathcal{D} \pi_\varsigma \succ_\mathcal{D} \pi_\varphi} \pi_\varsigma\Big) \cup \mathcal{D}_\varphi(\Pi).$$

From Proposition 3.5, we can derive that $\mathcal{D}' \rhd \mathcal{D}_\varphi(\Pi)$.

Now let us define

$$\mathcal{D}'' = \pi_\chi \cup \bigcup_{\pi_\chi \succ_\mathcal{D} \pi_\varsigma \succ_\mathcal{D} \pi_\varphi} \pi_\varsigma.$$

Hence,

$$\mathcal{D}_\chi(\Pi) = \mathcal{D}'' \cup \mathcal{D}_\varphi(\Pi).$$

$\mathcal{D}''$ is a union $\pi_{\alpha_1} \cup \pi_{\alpha_2} \cup \cdots \cup \pi_{\alpha_n}$ of modules such that $\pi_{\alpha_i} \succ_{\mathcal{D}} \pi_\varphi$, for $1 \leq i \leq n$, and

$$\mathcal{D}_\varphi(\Pi) = \pi_\varphi \cup \bigcup_{\pi_\varphi \succ_{\mathcal{D}} \pi_\zeta} \pi_\zeta$$

is a union $\pi_{\beta_1} \cup \pi_{\beta_2} \cup \cdots \cup \pi_{\beta_m}$ of modules such that $\pi_{\beta_j} = \pi_\varphi$ or $\pi_\varphi \succ_{\mathcal{D}} \pi_{\beta_j}$, for $1 \leq j \leq m$. By transitivity of $\succ_{\mathcal{D}}$, we have for $1 \leq i \leq n$, $1 \leq j \leq m$, that $\pi_{\alpha_i} \succ_{\mathcal{D}} \pi_{\beta_j}$, and since $\succ_{\mathcal{D}}$ is a subrelation of $\rhd$, also $\pi_{\alpha_i} \rhd \pi_{\beta_j}$ holds. By repeated application of Proposition 3.5.2, we get that $\pi_{\alpha_i} \rhd \mathcal{D}_\varphi(\Pi)$, for $1 \leq i \leq n$. From that, by repeated application of Proposition 3.5.1, we get that $\mathcal{D}'' \rhd \mathcal{D}_\varphi(\Pi)$.

Now, in view of the above, the condition

$$A_\mathcal{M} \in AS(\mathcal{D}_\chi(\Pi) \cup \mathcal{D}')$$

can be written as

$$A_\mathcal{M} \in AS(\mathcal{D}_\varphi(\Pi) \cup \mathcal{D}' \cup \mathcal{D}'').$$

Furthermore, from $\mathcal{D}' \rhd \mathcal{D}_\varphi(\Pi)$ and $\mathcal{D}'' \rhd \mathcal{D}_\varphi(\Pi)$, we get by Proposition 3.5 that for $\mathcal{D}''' = \mathcal{D}' \cup \mathcal{D}''$ it holds that $\mathcal{D}''' \rhd \mathcal{D}_\varphi(\Pi)$. Thus, we have

$$A_\mathcal{M} \in AS(\mathcal{D}_\varphi(\Pi) \cup \mathcal{D}''')$$

for some $\mathcal{D}''' \rhd \mathcal{D}_\varphi(\Pi)$. Therefore, by Definition 3.9, $A_\mathcal{M}$ is a φ-MAS for Π. ■

We need the following two lemmas for showing several properties of meta-answer-sets:

Lemma 3.11 *Let Π be a propositional disjunctive program over alphabet $\mathcal{A}$, $A_\mathcal{M}$ a set of atoms over $\mathcal{A}_\mathcal{M}(\mathcal{A})$, $\chi \in \mathcal{MOD}(\Pi)$, $\varphi \in \mathcal{MOD}(\Pi)$, and H_{π_χ} the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_\chi(\Pi))$, being a ground instance of a rule in π_χ. If $A_\mathcal{M}$ is both a χ-MAS and a φ-MAS for Π such that $\pi_\chi \succ_{\mathcal{D}} \pi_\varphi$, it holds that $A_\mathcal{M} \setminus H_{\pi_\chi}$ is a φ-MAS for Π.*

Proof. Assume $A_\mathcal{M}$ is both a χ-MAS and a φ-MAS for Π such that $\pi_\chi \succ_{\mathcal{D}} \pi_\varphi$. Then, by Definition 3.9 it holds that $A_\mathcal{M} \in AS(\mathcal{D}_\chi(\Pi) \cup \mathcal{D}')$ such that $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$. Therefore, we have $A_\mathcal{M} \in AS((\mathcal{D}_\chi(\Pi) \setminus \pi_\chi) \cup \pi_\chi \cup \mathcal{D}')$. As $(\pi_\chi \cup \mathcal{D}') \rhd (\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)$, we get by Proposition 3.6 that

$$A_\mathcal{M} \in \bigcup_{A' \in AS(\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)} AS(A' \cup \pi_\chi \cup \mathcal{D}').$$

Therefore, for some $A' \in AS(\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)$, it holds that $A_\mathcal{M} \in AS(A' \cup \pi_\chi \cup \mathcal{D}')$. As A' is a set of facts in $A' \cup \pi_\chi \cup \mathcal{D}'$, every answer set of $A' \cup \pi_\chi \cup \mathcal{D}'$ contains all atoms in A'. Thus, we have that

$$A_\mathcal{M} = A' \cup \mathcal{F}_{\pi_\chi} \cup \mathcal{F}_{\mathcal{D}'},$$

where $\mathcal{F}_{\pi_\chi} \subseteq H_{\pi_\chi}$ and $\mathcal{F}_{\mathcal{D}'}$ is a set of atoms such that every atom in $\mathcal{F}_{\mathcal{D}'}$ occurs in the head of a ground instance of some rule in $\mathcal{D}'$. Note that since $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$ holds, $\mathcal{F}_{\mathcal{D}'}$ does not contain any atom from $Gr(\mathcal{D}_\chi(\Pi))$, and thus $\mathcal{F}_{\mathcal{D}'}$ does not contain any ground instances of atoms from A' and $\mathcal{F}_{\pi_\chi}$. It holds that

$$A_\mathcal{M} \setminus H_{\pi_\chi} = A' \cup \mathcal{F}_{\mathcal{D}'},$$

and therefore also

$$A_\mathcal{M} \setminus H_{\pi_\chi} \in AS(A' \cup \mathcal{F}_{\mathcal{D}'}).$$

As $A' \in AS(\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)$, it holds that

$$A_\mathcal{M} \setminus H_{\pi_\chi} \in \bigcup_{A' \in AS(\mathcal{D}_\chi(\Pi)\setminus\pi_\chi)} AS(A' \cup \mathcal{F}_{\mathcal{D}'}).$$

As $\mathcal{F}_{\mathcal{D}'} \rhd (\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)$, we get by Proposition 3.6 that

$$A_\mathcal{M} \setminus H_{\pi_\chi} \in AS((\mathcal{D}_\chi(\Pi) \setminus \pi_\chi) \cup \mathcal{F}_{\mathcal{D}'}).$$

From $\pi_\chi \succ_\mathcal{D} \pi_\varphi$ we know that $\mathcal{D}_\chi(\Pi) \setminus \pi_\chi = \mathcal{D}_\varphi(\Pi) \cup M$, where M is a union of modules such that $M \rhd \mathcal{D}_\varphi(\Pi)$. It holds that

$$A_\mathcal{M} \setminus H_{\pi_\chi} \in AS(\mathcal{D}_\varphi(\Pi) \cup M \cup \mathcal{F}_{\mathcal{D}'}).$$

As $(M \cup \mathcal{F}_{\mathcal{D}'}) \rhd \mathcal{D}_\varphi(\Pi)$, by Definition 3.9, $A_\mathcal{M} \setminus H_{\pi_\chi}$ is a φ-MAS for Π. ■

Lemma 3.12 *Let Π be a propositional disjunctive program over alphabet $\mathcal{A}$, $A_\mathcal{M}$ a set of atoms over $\mathcal{A}_\mathcal{M}(\mathcal{A})$, $\chi \in \mathcal{MOD}(\Pi)$, and H_{π_χ} the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_\chi(\Pi))$, being a ground instance of a rule in π_χ. If $A_\mathcal{M}$ is a χ-MAS for Π, then it holds that $A_\mathcal{M} \in AS((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$.*

Proof. Assume $A_\mathcal{M}$ is a χ-MAS for Π. Then, by Definition 3.9 it holds that $A_\mathcal{M} \in AS(\mathcal{D}_\chi(\Pi) \cup \mathcal{D}')$ such that $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$. Thus, by Proposition 3.6, we get that

$$A_\mathcal{M} \in \bigcup_{A' \in AS(\mathcal{D}_\chi(\Pi))} AS(A' \cup \mathcal{D}').$$

Therefore, for some $A' \in AS(\mathcal{D}_\chi(\Pi))$, it holds that $A_\mathcal{M} \in AS(A' \cup \mathcal{D}')$. As A' is a set of facts in $A' \cup \mathcal{D}'$, every answer set of $A' \cup \mathcal{D}'$ contains all atoms in A'. Thus, we have that

$$A_\mathcal{M} = A' \cup \mathcal{F},$$

where $\mathcal{F}$ is a set of atoms such that every atom in $\mathcal{F}$ occurs in the head of a ground instance of some rule in $\mathcal{D}'$. Note that since $\mathcal{D}' \rhd \mathcal{D}_\chi(\Pi)$ holds, $\mathcal{F}$ does not contain any atom from $Gr(\mathcal{D}_\chi(\Pi))$, and thus $\mathcal{F}$ does not contain any ground instances of atoms from π_χ.

As $(A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi$ is an absolutely tight program, for showing that $A_\mathcal{M} \in AS((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$, by Theorem 2.34 it suffices to show that $A_\mathcal{M}$ satisfies $Gr((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$ and $A_\mathcal{M}$ is supported by $Gr((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$. Clearly, $A_\mathcal{M}$ satisfies every fact in $A_\mathcal{M} \setminus H_{\pi_\chi}$. Consider a rule $r_\mathcal{M}$ in $Gr((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$, being a ground instance of some rule in π_χ. Assume that $r_\mathcal{M}$ is applicable under $A_\mathcal{M}$. Then, it holds that $r_\mathcal{M}$ is also applicable under A'. As $r_\mathcal{M}$ is a rule in $Gr(\mathcal{D}_\chi(\Pi))$ and A' is an answer set of $\mathcal{D}_\chi(\Pi)$, it holds that $head(r_\mathcal{M}) \cap A' \neq \emptyset$. Hence, we have that $head(r_\mathcal{M}) \cap A_\mathcal{M} \neq \emptyset$, and therefore $A_\mathcal{M}$ satisfies $r_\mathcal{M}$.

As for every $a \in A_\mathcal{M} \setminus H_{\pi_\chi}$, $a \leftarrow$ is a fact in $Gr((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$, $\{a\}$ is supported by $Gr((A_\mathcal{M} \setminus H_{\pi_\chi}) \cup \pi_\chi)$ with respect to $A_\mathcal{M}$. Consider an arbitrary atom $a \in A_\mathcal{M} \cap H_{\pi_\chi}$. Then, it holds that $a \in A'$. Since A' is an answer set of $\mathcal{D}_\chi(\Pi)$, $\{a\}$ must be supported by $\mathcal{D}_\chi(\Pi)$ with respect to A'. Moreover, since $a \in H_{\pi_\chi}$, a rule from $\mathcal{D}_\chi(\Pi)$ supporting $\{a\}$ must be a ground instance of a rule in π_χ, as $\pi_\chi \rhd (\mathcal{D}_\chi(\Pi) \setminus \pi_\chi)$. Hence, there must be a ground instance $r_\mathcal{M}$ of a rule in π_χ such that $head(r_\mathcal{M}) \cap A' = \{a\}$ and $r_\mathcal{M}$ is applicable under A'. Then, since $A_\mathcal{M} = A' \cup \mathcal{F}$ and $\mathcal{F}$ does not contain any ground instances of atoms from π_χ,

also $head(r_{\mathcal{M}}) \cap A_{\mathcal{M}} = \{a\}$. Furthermore, $r_{\mathcal{M}}$ is also a rule in $Gr((A_{\mathcal{M}} \setminus H_{\pi_\chi}) \cup \pi_\chi)$, being applicable under $A_{\mathcal{M}}$. Thus, $\{a\}$ is supported by $Gr((A_{\mathcal{M}} \setminus H_{\pi_\chi}) \cup \pi_\chi)$ with respect to $A_{\mathcal{M}}$.

As for every atom $a \in A_{\mathcal{M}}$, $\{a\}$ is supported by $Gr((A_{\mathcal{M}} \setminus H_{\pi_\chi}) \cup \pi_\chi)$ with respect to $A_{\mathcal{M}}$, by Proposition 2.31, $A_{\mathcal{M}}$ is supported by $Gr((A_{\mathcal{M}} \setminus H_{\pi_\chi}) \cup \pi_\chi)$.

Hence, $A_{\mathcal{M}}$ is an answer set of $((A_{\mathcal{M}} \setminus H_{\pi_\chi}) \setminus H_{\pi_\chi}) \cup \pi_\chi$. ■

In the following subsections, we will describe the modules of the meta-program in detail.

3.3.5 Transformation to the Meta-Level

The input for the debugging system is an encoding $\pi_{in(\Pi)}$ of a program Π to debug.

Definition 3.13 Let Π be a propositional disjunctive logic program. Then, the program $\pi_{in(\Pi)}$ is given by the following set of rules:

$$\begin{aligned}
\{head(l_r, l_a) \leftarrow \quad & | \; a \in head(r) \text{ and } r \in \Pi\} \cup \\
\{bodyP(l_r, l_a) \leftarrow \quad & | \; a \in body^+(r) \text{ and } r \in \Pi\} \cup \\
\{bodyN(l_r, l_a) \leftarrow \quad & | \; a \in body^-(r) \text{ and } r \in \Pi\}.
\end{aligned}$$ □

Having Π encoded this way, we can reason about its properties at the meta-level.

Example 3.14 Consider the following program:

$$\begin{aligned}
\Pi_{ex} = \{\; r_1 &= a \vee b \leftarrow c, not\ d, \\
r_2 &= c \leftarrow d, \\
r_3 &= d \vee e \leftarrow \;\}.
\end{aligned}$$

The encoding $\pi_{in(\Pi_{ex})}$ for Π_{ex} is then as follows:

$$\begin{aligned}
\pi_{in(\Pi_{ex})} = \{\; & head(l_{r_1}, l_a) \leftarrow, \\
& head(l_{r_1}, l_b) \leftarrow, \\
& bodyP(l_{r_1}, l_c) \leftarrow, \\
& bodyN(l_{r_1}, l_d) \leftarrow, \\
& head(l_{r_2}, l_c) \leftarrow, \\
& bodyP(l_{r_2}, l_d) \leftarrow, \\
& head(l_{r_3}, l_c) \leftarrow, \\
& head(l_{r_3}, l_e) \leftarrow \;\}.
\end{aligned}$$ ♢

The following two results are obvious.

Proposition 3.15 *For every propositional program* Π*, it holds that*

$$AS(\pi_{in(\Pi)}) = \{\{a \mid \quad a \leftarrow \; \in \pi_{in(\Pi)}\}\}.$$

Proposition 3.16 *Let $A_{\mathcal{M}}$ be the unique in(Π)-MAS for a propositional disjunctive program* Π. *Then, it holds that*

$$\begin{aligned} &head(l_r, l_a) \;\; \in A_{\mathcal{M}} \text{ iff } a \in head(r), \\ &bodyP(l_r, l_a) \in A_{\mathcal{M}} \text{ iff } a \in body^+(r), \quad \text{and} \\ &bodyN(l_r, l_a) \in A_{\mathcal{M}} \text{ iff } a \in body^-(r). \end{aligned}$$

For the program Π_{ex} from Example 3.14 we have that

$$\begin{aligned} AS(\pi_{in(\Pi_{ex})}) = \{\{ &head(l_{r_1}, l_a), head(l_{r_1}, l_b), bodyP(l_{r_1}, l_c), bodyN(l_{r_1}, l_d), \\ &head(l_{r_2}, l_c), bodyP(l_{r_2}, l_d), \\ &head(l_{r_3}, l_c), head(l_{r_3}, l_e)\}\}. \end{aligned}$$

3.3.6 Auxiliary Rules

Subprogram π_{aux} consists of auxiliary meta-rules which are used to identify all rules and atoms occurring in Π:

Definition 3.17 Module π_{aux} consists of the following rules:

$$atom(A) \leftarrow head(\_, A), \tag{3}$$
$$atom(A) \leftarrow bodyP(\_, A), \tag{4}$$
$$atom(A) \leftarrow bodyN(\_, A), \tag{5}$$
$$rule(R) \leftarrow head(R, \_), \tag{6}$$
$$rule(R) \leftarrow bodyP(R, \_), \tag{7}$$
$$rule(R) \leftarrow bodyN(R, \_). \tag{8}$$

□

Since $\mathcal{D}_{aux}(\Pi)$ is normal and stratified, we know by Proposition 2.39 that it has a unique answer set A_{aux} Informally speaking, A_{aux} extends the unique answer set of $\mathcal{D}_{in}(\Pi)$ by meta-atoms indicating which rules and atoms are contained in Π.

Proposition 3.18 *Let $A_{\mathcal{M}}$ be an aux-MAS for a propositional disjunctive program* Π. *Then, it holds that*

$$\begin{aligned} &atom(l_a) \in A_{\mathcal{M}} \text{ iff } a \in HB(\Pi) \quad \text{and} \\ &rule(l_r) \;\; \in A_{\mathcal{M}} \text{ iff } r \in \Pi. \end{aligned}$$

Proof. Part 1: ($\Rightarrow$) Assume that $atom(l_a) \in A_{\mathcal{M}}$ holds. Therefore, there must be an applicable rule in $Gr(\mathcal{D}_{aux}(\Pi))$ such that $atom(l_a)$ is in the head of this rule. Only ground instances of rules (3), (4), and (5) can satisfy this property. Thus, for some l_r, either $head(l_r, l_a) \in A_{\mathcal{M}}$, $bodyP(l_r, l_a) \in A_{\mathcal{M}}$, or $bodyN(l_r, l_a) \in A_{\mathcal{M}}$. From that, we know by Proposition 3.16 that a is an atom in the head, in the positive, or in the negative body of some rule $r \in \Pi$, and hence $a \in HB(\Pi)$.

($\Leftarrow$) Assume that $a \in HB(\Pi)$ holds. Therefore, a must be in the head, in the positive, or in the negative body of some rule $r \in \Pi$. From that, we know by Proposition 3.16 that either $head(l_r, l_a) \in A_{\mathcal{M}}$, $bodyP(l_r, l_a) \in A_{\mathcal{M}}$, or $bodyN(l_r, l_a) \in A_{\mathcal{M}}$. Thus, there must be

an applicable ground instance of one of the rules (3), (4), and (5) in $Gr(\mathcal{D}_{aux}(\Pi))$ such that $atom(l_a)$ is in the head of this rule. Therefore, we have that $atom(l_a) \in A_{\mathcal{M}}$.

Part 2: ($\Rightarrow$) Assume that $rule(l_r) \in A_{\mathcal{M}}$ holds. Therefore, there must be an applicable rule in $Gr(\mathcal{D}_{aux}(\Pi))$ such that $rule(l_r)$ is in the head of this rule. Only ground instances of rules (6), (7), and (8) can satisfy this property. Thus, for some l_a, either $head(l_r, l_a) \in A_{\mathcal{M}}$, $bodyP(l_r, l_a) \in A_{\mathcal{M}}$, or $bodyN(l_r, l_a) \in A_{\mathcal{M}}$. From that, we know by Proposition 3.16 that a is an atom in the head or in the body of rule $r \in \Pi$, and hence $r \in \Pi$.

($\Leftarrow$) Assume rule $r \in \Pi$. By definition of a rule, there must be an atom a in either the head, in the positive or in the negative body of r. From that, we know by Proposition 3.16 that either $head(l_r, l_a) \in A_{\mathcal{M}}$, $bodyP(l_r, l_a) \in A_{\mathcal{M}}$, or $bodyN(l_r, l_a) \in A_{\mathcal{M}}$. Thus, there must be an applicable ground instance of one of the rules (6), (7), and (8) in $Gr(\mathcal{D}_{aux}(\Pi))$ such that $rule(l_r)$ is in the head of this rule. Therefore, we have that $rule(l_r) \in A_{\mathcal{M}}$. ■

3.3.7 Guessing an Interpretation

The purpose of module π_{int}, described next, is guessing an interpretation I for the program Π to debug. This is done because each answer set of meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ is used to investigate the outcome of Π, under a particular interpretation.

Definition 3.19 Let Π be a propositional disjunctive program. Then, π_{int} consists of the following two rules:

$$int(A) \leftarrow atom(A), not\ \overline{int}(A), \tag{9}$$

$$\overline{int}(A) \leftarrow atom(A), not\ int(A). \tag{10}$$

□

Intuitively, these two rules partition the atoms occurring in Π into two categories such that they either belong to an interpretation or not.

For associating meta-answer-sets with interpretations for Π, we introduce the notion of meta-answer-sets guessing an interpretation:

Definition 3.20 Let $A_{\mathcal{M}}$ be a meta-answer-set for a propositional disjunctive program Π such that $A_{\mathcal{M}}$ is an *int*-MAS for Π. We say that $A_{\mathcal{M}}$ *guesses interpretation* I precisely when I is an interpretation for Π and, for all $a \in HB(\Pi)$, $int(l_a) \in A_{int}$ iff $a \in I$. □

Proposition 3.21 *Let I be an interpretation for a propositional disjunctive program Π. Then, there is an $A_{int} \in AS(\mathcal{D}_{int}(\Pi))$ such that, for all $a \in HB(\Pi)$, $int(l_a) \in A_{int}$ iff $a \in I$.*

Proof. Since I is an interpretation for Π, we have $a \in HB(\Pi)$ for all $a \in I$, we know by Proposition 3.18 that $atom(l_a) \in A_{aux}$, where A_{aux} is the unique answer set of $\mathcal{D}_{aux}(\Pi)$. From that and $\pi_{int} \rhd \mathcal{D}_{aux}(\Pi)$, by Proposition 3.6, we can conclude that $AS(\mathcal{D}_{int}(\Pi)) = AS(\mathcal{D}_{aux}(\Pi) \cup \pi_{int}) = AS(A_{aux} \cup \pi_{int})$.

Consider $A_{int} = A_{aux} \cup \{int(l_a) \mid a \in I\} \cup \{\overline{int}(l_a) \mid a \in HB(\Pi) \setminus I\}$. Note that A_{int} meets the condition that $int(l_a) \in A_{int}$ iff $a \in I$. We show that $A_{int} \in AS(A_{aux} \cup \pi_{int})$.

First, note that $Gr(A_{aux} \cup \pi_{int})$ is an absolutely tight program, since the atoms of the facts in A_{aux} have no outgoing edges in the positive dependency graph G of $Gr(A_{aux} \cup \pi_{int})$, and the ground instances of the rules in π_{int} only have edges to atoms in A_{aux} in G.

In order to show that $A_{int} \in AS(A_{aux} \cup \pi_{int})$, since $Gr(A_{aux} \cup \pi_{int})$ is an absolutely tight program, by Theorem 2.34 it suffices to verify that A_{int} satisfies $Gr(A_{aux} \cup \pi_{int})$ and A_{int} is supported by $Gr(A_{aux} \cup \pi_{int})$.

Since $A_{aux} \subseteq A_{int}$, all ground instances of rules from A_{aux} are satisfied by A_{int}. We will now consider the remaining rules in $Gr(A_{aux} \cup \pi_{int})$, namely the ground instances of the rules (9) and (10).

Consider the rule

$$r_{\mathcal{M}} \;=\; int(l_a) \leftarrow atom(l_a), not\ \overline{int}(l_a)\,,$$

being a ground instance of rule (9). Assume that $A_{int} \models body(r_{\mathcal{M}})$. We have that $atom(l_a) \in A_{int}$ and $\overline{int}(l_a) \notin A_{int}$. From $atom(l_a) \in A_{int}$ and since A_{int} is an *aux*-MAS for Π, by Proposition 3.18, we get that $a \in HB(\Pi)$. From that and $\overline{int}(l_a) \notin A_{int}$, we know by definition of A_{int} that $a \in I$. Hence, also by definition of A_{int}, we have that $int(l_a) \in A_{int}$. We showed that each ground instance of rule (9) is satisfied by A_{int}.

Now consider

$$r_{\mathcal{M}} \;=\; \overline{int}(l_a) \leftarrow atom(l_a), not\ int(l_a)\,,$$

being a ground instance of rule (10). Assume that $A_{int} \models body(r_{\mathcal{M}})$. We have that $atom(l_a) \in A_{int}$ and $int(l_a) \notin A_{int}$. From $atom(l_a) \in A_{int}$ and since A_{int} is an *aux*-MAS for Π, by Proposition 3.18, we get that $a \in HB(\Pi)$. From $int(l_a) \notin A_{int}$, we know by definition of A_{int} that $a \notin I$. Therefore, also by definition of A_{int}, we have that $\overline{int}(l_a) \in A_{int}$. We showed that each ground instance of rule (9) is satisfied by A_{int}. We have that A_{int} satisfies $Gr(A_{aux} \cup \pi_{int})$ and will now show that A_{int} is supported by $Gr(A_{aux} \cup \pi_{int})$. Since $Gr(A_{aux} \cup \pi_{int})$ is non-disjunctive, by Proposition 2.31 it is sufficient to show that for every $a_{\mathcal{M}} \in A_{int}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{aux} \cup \pi_{int})$ with respect to A_{int}.

For all atoms $a_{\mathcal{M}} \in A_{int}$, we have either $a_{\mathcal{M}} \in A_{aux}$, $a_{\mathcal{M}} \in \{int(l_a) \mid a \in I\}$ or $a_{\mathcal{M}} \in \{\overline{int}(l_a) \mid a \in HB(\Pi) \setminus I\}$. In case $a_{\mathcal{M}} \in A_{aux}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{aux} \cup \pi_{int})$ with respect to A_{int}, because $a_{\mathcal{M}} \leftarrow$ is a fact in $A_{aux} \cup \pi_{int}$.

If $a_{\mathcal{M}} = int(l_a)$, for some $a \in I$, consider the rule

$$r_{\mathcal{M}} \;=\; int(l_a) \leftarrow atom(l_a), not\ \overline{int}(l_a)\,,$$

being a ground instance of rule (9). Since $a \in I$, we have that $a \in HB(\Pi)$, and since A_{int} is an *aux*-MAS for Π, by Proposition 3.18, we get that $atom(l_a) \in A_{int}$. From $a \in I$ we can conclude by definition of A_{int} that $\overline{int}(l_a) \notin A_{int}$. Hence, $A_{int} \models body(r_{\mathcal{M}})$, and therefore, for all $a_{\mathcal{M}} \in \{int(l_a) \mid a \in I\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{aux} \cup \pi_{int})$ with respect to A_{int}.

Finally, if $a_{\mathcal{M}} = \overline{int}(l_a)$, for some $a \in HB(\Pi) \setminus I$, consider the rule

$$r_{\mathcal{M}} \;=\; \overline{int}(l_a) \leftarrow atom(l_a), not\ int(l_a)\,,$$

being a ground instance of rule (10). Since $a \in HB(\Pi) \setminus I$ and A_{int} is an *aux*-MAS for Π, by Proposition 3.18, we get that $atom(l_a) \in A_{int}$. From $a \notin I$ we can conclude by definition of A_{int} that $int(l_a) \notin A_{int}$. Hence, $A_{int} \models body(r_{\mathcal{M}})$, and therefore, for all $a_{\mathcal{M}} \in \{\overline{int}(l_a) \mid a \in HB(\Pi) \setminus I\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{aux} \cup \pi_{int})$ with respect to A_{int}.

We showed that A_{int} is supported by $Gr(A_{aux} \cup \pi_{int})$. Since A_{int} satisfies $Gr(A_{aux} \cup \pi_{int})$, A_{int} is supported by $Gr(A_{aux} \cup \pi_{int})$, and $Gr(A_{aux} \cup \pi_{int})$ is an absolutely tight program, by Theorem 2.34, we have that $A_{int} \in AS(Gr(A_{aux} \cup \pi_{int}))$, and therefore also $A_{int} \in AS(\mathcal{D}_{int}(\Pi))$ holds. ∎

The next two corollaries follow immediately from Proposition 3.21 and Definition 3.20:

Corollary 3.22 *Let I be an interpretation for a propositional disjunctive program Π. Then, there is an $A_{int} \in AS(\mathcal{D}_{int}(\Pi))$ such that A_{int} is an int-MAS for Π guessing I.*

Corollary 3.23 *Let Π be a propositional disjunctive program and A_{int} an int-MAS for Π. Then, there is an interpretation I for Π such that A_{int} guesses I.*

3.3.8 Rule Applicability

The purpose of module π_{ap} is extending an *int*-MAS for Π guessing interpretation I with meta-atoms stating which rules of Π are applicable or blocked under I.

Definition 3.24 Module π_{int} consists of the following rules:

$$ap(R) \leftarrow not\ bl(R), rule(R), \tag{11}$$

$$bl(R) \leftarrow bodyN(R, A), int(A), \tag{12}$$

$$bl(R) \leftarrow bodyP(R, A), not\ int(A). \tag{13}$$

□

Lemma 3.25 *Let $A_{\mathcal{M}}$ be an int-MAS for a propositional disjunctive program Π guessing interpretation I such that $\pi_{ap} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{ap}$ has a unique answer set A_{ap} such that $ap(l_r) \in A_{ap}$ (respectively $bl(l_r) \in A_{ap}$) iff r is an applicable (respectively blocked) rule in Π under I.*

Proof. Since $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ is normal and stratified program, we know by Proposition 2.39 that $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ has a single answer set, and hence also $A_{\mathcal{M}} \cup \pi_{ap}$ has a single answer set.

Consider the set $A_{ap} = A_{\mathcal{M}} \cup \{ap(l_r) \mid r \in \mathsf{GEN}(\Pi, I)\} \cup \{bl(l_r) \mid r \in \Pi \setminus \mathsf{GEN}(\Pi, I)\}$. Note that since $A_{\mathcal{M}}$ is an *int*-MAS for Π guessing I, and $(A_{ap} \setminus A_{\mathcal{M}}) \rhd A_{\mathcal{M}}$, also A_{ap} is an *int*-MAS for Π guessing I. From the definition of A_{ap}, we get the property that $ap(l_r) \in A_{ap}$ (respectively $bl(l_r) \in A_{ap}$) holds iff r is applicable (respectively blocked) in Π under I.

The remaining task is showing that A_{ap} is the answer set of $A_{\mathcal{M}} \cup \pi_{ap}$. Since $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ is absolutely tight, by Theorem 2.34, it suffices to show that A_{ap} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ and A_{ap} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$.

Since $A_{\mathcal{M}} \subseteq A_{ap}$, the facts of $A_{\mathcal{M}}$ are satisfied by A_{ap}. The remaining rules of $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ are the ground instances of the rules (11), (12), and (13).

First, consider a rule

$$r_{\mathcal{M}} \ = \ ap(l_r) \leftarrow not\ bl(l_r), rule(l_r),$$

being a ground instance of rule (11). Assume that $A_{ap} \models body(r_{\mathcal{M}})$ but also $ap(l_r) \notin A_{ap}$ hold. Thus, $bl(l_r) \notin A_{ap}$ and $rule(l_r) \in A_{ap}$. Since A_{ap} is an *aux*-MAS for Π, by Proposition 3.18, r is a rule of Π. By definition of A_{ap} and since $bl(l_r) \notin A_{ap}$, we get that $r \in \mathsf{GEN}(\Pi, I)$. Therefore, by definition of A_{ap}, we have $ap(l_r) \in A_{ap}$ which contradicts the assumption that $ap(l_r) \notin A_{ap}$. Thus, ground instances of rule (11) are satisfied by A_{ap}.

Now, consider a rule

$$r_{\mathcal{M}} \ = \ bl(l_r) \leftarrow bodyN(l_r, l_a), int(l_a),$$

being a ground instance of rule (12). Assume that $A_{ap} \models body(r_{\mathcal{M}})$ but also $bl(l_r) \notin A_{ap}$ hold. Thus, $bodyN(l_r, l_a), int(l_a) \in A_{ap}$. Since A_{ap} is an *int*-MAS for Π guessing I, we know

that $a \in I$, and by Proposition 3.18, r is a rule of Π. Furthermore, $a \in \mathit{body}^-(\Pi)$. Rule r cannot be applicable under I since $a \in \mathit{body}^-(\Pi)$ but $a \in I$. Thus, we have $r \in \Pi \setminus \mathsf{GEN}(\Pi, I)$. Therefore, by definition of A_{ap}, we have $\mathit{bl}(l_r) \in A_{ap}$ which contradicts the assumption that $\mathit{bl}(l_r) \notin A_{ap}$. Thus, ground instances of rule (12) are satisfied by A_{ap}.

Finally, consider a rule

$$r_{\mathcal{M}} \;=\; \mathit{bl}(l_r) \leftarrow \mathit{bodyP}(l_r, l_a), \mathit{not}\ \mathit{int}(l_a),$$

being a ground instance of rule (13). Assume that $A_{ap} \models \mathit{body}(r_{\mathcal{M}})$ but also $\mathit{bl}(l_r) \notin A_{ap}$ hold. Thus, $\mathit{bodyP}(l_r, l_a) \in A_{ap}$ and $\mathit{int}(l_a) \notin A_{ap}$. Since A_{ap} is an *int*-MAS for Π guessing I, we know that $a \notin I$, and by Proposition 3.18, r is a rule of Π. Furthermore, $a \in \mathit{body}^+(\Pi)$. Rule r cannot be applicable under I since $a \in \mathit{body}^+(\Pi)$ but $a \notin I$. Thus, we have $r \in \Pi \setminus \mathsf{GEN}(\Pi, I)$. Therefore, by definition of A_{ap}, we have $\mathit{bl}(l_r) \in A_{ap}$ which contradicts the assumption that $\mathit{bl}(l_r) \notin A_{ap}$. Thus, ground instances of rule (13) are satisfied by A_{ap}. Therefore, we have shown that A_{ap} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{ap})$.

We now show that A_{ap} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$. Since $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ is non-disjunctive, by Proposition 2.31 it is sufficient to show that for every $a_{\mathcal{M}} \in A_{ap}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ with respect to A_{ap}.

Clearly, for all atoms $a_{\mathcal{M}} \in A_{ap}$, we have either $a_{\mathcal{M}} \in A_{\mathcal{M}}$, $a_{\mathcal{M}} \in \{\mathit{ap}(l_r) \mid r \in \mathsf{GEN}(\Pi, I)\}$, or $a_{\mathcal{M}} \in \{\mathit{bl}(l_r) \mid r \in \Pi \setminus \mathsf{GEN}(\Pi, I)\}$. In case $a_{\mathcal{M}} \in A_{\mathcal{M}}$, $\{a_{\mathcal{M}}\}$ is trivially supported by $A_{\mathcal{M}} \cup \pi_{ap}$ with respect to A_{ap}, because $a_{\mathcal{M}}$ is a fact in $A_{\mathcal{M}} \cup \pi_{ap}$.

Assume that$a_{\mathcal{M}} \in \{\mathit{ap}(l_r) \mid r \in \mathsf{GEN}(\Pi, I)\}$ holds. Then, it holds that $a_{\mathcal{M}} = \mathit{ap}(l_r)$, where r is a rule of Π, applicable under I. Thus, by definition of A_{ap}, $\mathit{bl}(l_r) \notin A_{ap}$, and since A_{ap} is an *aux*-MAS for Π, by Proposition 3.18, we have that $\mathit{rule}(l_r) \in A_{ap}$. Now, for rule

$$r_{\mathcal{M}} \;=\; \mathit{ap}(l_r) \leftarrow \mathit{not}\ \mathit{bl}(l_r), \mathit{rule}(l_r),$$

being a ground instance of rule (11), we have that $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ap}), \mathit{head}(r_{\mathcal{M}}) = \{a_{\mathcal{M}}\}$ and $A_{ap} \models \mathit{body}(r_{\mathcal{M}})$. Thus, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ with respect to A_{ap}.

Now, assume that $a_{\mathcal{M}} \in \{\mathit{bl}(l_r) \mid r \in \Pi \setminus \mathsf{GEN}(\Pi, I)\}$. Then, it holds that $a_{\mathcal{M}} = \mathit{bl}(l_r)$, where r is a rule of Π, blocked under I. Thus, there must be some $a \in \mathit{body}(r)$ such that either $a \in \mathit{body}^-(r)$ but $a \in I$ or $a \in \mathit{body}^+(r)$ but $a \notin I$. Consider the first case, $a \in \mathit{body}^-(r)$ but $a \in I$. Since A_{ap} is an *int*-MAS for Π guessing I, we know that $\mathit{int}(l_a) \in A_{ap}$ and, by Proposition 3.18, we have $\mathit{bodyN}(l_r, l_a) \in A_{ap}$. Therefore, for rule

$$r_{\mathcal{M}} \;=\; \mathit{bl}(l_r) \leftarrow \mathit{bodyN}(l_r, l_a), \mathit{int}(l_a),$$

being a ground instance of rule (12), we have that $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ap}), \{a_{\mathcal{M}}\} = \mathit{head}(r_{\mathcal{M}})$, and $A_{ap} \models \mathit{body}(r_{\mathcal{M}})$. Thus, $\{a_{\mathcal{M}}\}$ is supported by $A_{\mathcal{M}} \cup \pi_{ap}$ with respect to A_{ap}.

It remains to consider the second case, $a \in \mathit{body}^+(r)$ but $a \notin I$. Since A_{ap} is an *int*-MAS for Π guessing I, we know that $\mathit{int}(l_a) \notin A_{ap}$ and, by Proposition 3.18, we have $\mathit{bodyP}(l_r, l_a) \in A_{ap}$. Therefore, for rule

$$r_{\mathcal{M}} \;=\; \mathit{bl}(l_r) \leftarrow \mathit{bodyP}(l_r, l_a), \mathit{not}\ \mathit{int}(l_a),$$

being a ground instance of rule (12), we have that $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ap}), \{a_{\mathcal{M}}\} = \mathit{head}(r_{\mathcal{M}})$, and $A_{ap} \models \mathit{body}(r_{\mathcal{M}})$. Thus, $\{a_{\mathcal{M}}\}$ is again supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$ with respect to A_{ap}.

Summarising, for all atoms $a_{\mathcal{M}} \in A_{ap}$, $\{a_{\mathcal{M}}\}$ is supported by $A_{\mathcal{M}} \cup \pi_{ap}$ with respect to A_{ap}, and therefore A_{ap} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ap})$. ■

Proposition 3.26 *Let A_{ap} be an ap-MAS for a propositional disjunctive program Π guessing interpretation I. Then, $ap(l_r) \in A_{ap}$ (respectively $bl(l_r) \in A_{ap}$) iff r is an applicable (respectively blocked) rule in Π under I.*

Proof. Let $H_{\pi_{ap}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{ap}(\Pi))$, being a ground instance of a rule in π_{ap}. By Lemma 3.12, it holds that $A_{ap} \in AS((A_{ap} \setminus H_{\pi_{ap}}) \cup \pi_{ap})$. Furthermore, by Proposition 3.10, A_{ap} is an *int*-MAS for Π. As A_{ap} is an *ap*-MAS, an *int*-MAS for Π, and $\pi_{ap} \succ_{\mathcal{D}} \pi_{int}$, by Lemma 3.11, we have that $A_{ap} \setminus H_{\pi_{ap}}$ is an *int*-MAS for Π. Note that $A_{ap} \setminus H_{\pi_{ap}}$ guesses I. From that, the fact that $\pi_{ap} \rhd (A_{ap} \setminus H_{\pi_{ap}})$, and since $A_{ap} \in AS((A_{ap} \setminus H_{\pi_{ap}}) \cup \pi_{ap})$, we conclude by Lemma 3.25 that $ap(l_r) \in A_{ap}$ (respectively $bl(l_r) \in A_{ap}$) iff r is an applicable (respectively blocked) rule in Π under I. ■

3.3.9 Dependency Graph

Module π_{dpcy} detects paths within the positive dependency graph of program Π. This is needed for reasoning about loops and their external support.

Definition 3.27 Module π_{dpcy} consists of the following rules:

$$dpcy(A, A) \leftarrow atom(A), \tag{14}$$

$$dpcy(A1, A2) \leftarrow head(R, A1), bodyP(R, A2), \tag{15}$$

$$dpcy(A1, A2) \leftarrow dpcy(A1, A3), dpcy(A3, A2), \tag{16}$$

$$strCon(A1, A2) \leftarrow dpcy(A1, A2), dpcy(A2, A1). \tag{17}$$

□

Lemma 3.28 *Let $A_{\mathcal{M}}$ be an aux-MAS for a propositional disjunctive program Π such that $\pi_{dpcy} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{dpcy}$ has a unique answer set, A_{dpcy}. Moreover, it holds that $dpcy(l_a, l_b) \in A_{dpcy}$ iff there is a path from a to b in the positive dependency graph of Π.*

Proof. That $A_{\mathcal{M}} \cup \pi_{dpcy}$ has a unique answer set, A_{dpcy}, follows from Proposition 2.39 since $A_{\mathcal{M}} \cup \pi_{dpcy}$ is normal and stratified. It remains to show that $dpcy(l_a, l_b) \in A_{dpcy}$ iff there is a path from a to b in the positive dependency graph of Π. Let G be the positive dependency graph of Π.

($\Rightarrow$) Let Δ be the set of all meta-atoms $dpcy(l_x, l_y) \in A_{dpcy}$ such that there is no path from x to y in G. We will show that $\Delta = \emptyset$. To this end, consider $A_\Delta = A_{dpcy} \setminus \Delta$. Since $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$ is a positive program, it coincides with its reduct $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})^{A_{dpcy}}$. Furthermore, since A_{dpcy} is an answer set of $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$ and A_Δ is a subset of A_{dpcy}, by showing that A_Δ satisfies $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$, from the minimality of answer sets, we get that $A_\Delta = A_{dpcy}$, and hence $\Delta = \emptyset$. It remains to show that A_Δ satisfies $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$.

Consider the facts in $A_{\mathcal{M}}$. Since $\pi_{dpcy} \rhd A_{\mathcal{M}}$, there is no meta-atom $dpcy(l_x, l_y)$ in $A_{\mathcal{M}}$. Therefore, A_Δ satisfies $A_{\mathcal{M}}$. The remaining rules of $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$ are the ground instances of the rules (14), (15), (16) and (17).

First consider some rule

$$r_{\mathcal{M}} \;=\; dpcy(l_a, l_a) \leftarrow atom(l_a),$$

being a ground instance of rule (14), and assume that $A_\Delta \models body(r_{\mathcal{M}})$. Since $A_\Delta \subseteq A_{dpcy}$, also $A_{dpcy} \models body(r_{\mathcal{M}})$, and thus $dpcy(l_a, l_a) \in A_{dpcy}$. Since A_Δ is an *aux*-MAS for Π, by

Proposition 3.18, we have $a \in HB(\Pi)$, and thus there is a path from a to a in G. Therefore, since $dpcy(l_a, l_a) \in A_{dpcy}$, we also have $dpcy(l_a, l_a) \in A_\Delta$. Consequently, ground instances of rule (14) are satisfied by A_Δ.

Now consider some rule

$$r_\mathcal{M} \;=\; dpcy(l_a, l_b) \leftarrow head(l_r, l_a), bodyP(l_r, l_b)\,,$$

being a ground instance of rule (15), and assume that $A_\Delta \models body(r_\mathcal{M})$. Since $A_\Delta \subseteq A_{dpcy}$, also $A_{dpcy} \models body(r_\mathcal{M})$ and thus $dpcy(l_a, l_b) \in A_{dpcy}$. Since A_Δ is an *aux*-MAS for Π, by Proposition 3.18, we have $a \in head(r), b \in body^+(r)$ for rule $r \in \Pi$, and thus there is a path from a to b in G. Therefore, since $dpcy(l_a, l_b) \in A_{dpcy}$, we also have $dpcy(l_a, l_b) \in A_\Delta$. Consequently, ground instances of rule (15) are satisfied by A_Δ.

Consider rule

$$r_\mathcal{M} \;=\; dpcy(l_a, l_b) \leftarrow dpcy(l_a, l_c), dpcy(l_c, l_b)\,,$$

being a ground instance of rule (16), and assume that $A_\Delta \models body(r_\mathcal{M})$. Since $A_\Delta \subseteq A_{dpcy}$, also $A_{dpcy} \models body(r_\mathcal{M})$, and thus $dpcy(l_a, l_b) \in A_{dpcy}$. From $dpcy(l_a, l_c)$, $dpcy(l_c, l_b) \in A_\Delta$ we know that there are paths from a to c and from c to b in G and thus we can conclude that there is also a path from a to b. Therefore, since $dpcy(l_a, l_b) \in A_{dpcy}$, we also have $dpcy(l_a, l_b) \in A_\Delta$. Consequently, ground instances of rule (16) are satisfied by A_Δ.

Finally, consider some rule

$$r_\mathcal{M} \;=\; strCon(l_a, l_b) \leftarrow dpcy(l_a, l_b), dpcy(l_b, l_a)\,,$$

being a ground instance of rule (17), and assume that $A_\Delta \models body(r_\mathcal{M})$. Since $A_\Delta \subseteq A_{dpcy}$, also $A_{dpcy} \models body(r_\mathcal{M})$, and thus $strCon(l_a, l_b) \in A_{dpcy}$. Since $strCon(l_a, l_b) \notin \Delta$, it is also contained in A_Δ. Therefore, ground instances of rule (17) are satisfied by A_Δ.

Consequently, A_Δ satisfies the reduct $Gr(A_\mathcal{M} \cup \pi_{dpcy})^{A_{dpcy}}$. This completes the proof that A_Δ satisfies $Gr(A_\mathcal{M} \cup \pi_{dpcy})$.

($\Leftarrow$) The proof that a path from atom a to atom a_n in G implies that $dpcy(l_a, l_{a_n}) \in A_{dpcy}$, is done by induction on the length of the path.

Base case: Let $\Psi = a$ be an arbitrary path of length 0 in G. Then, $a \in HB(\Pi)$ is an atom in Π. Since A_{dpcy} is an *aux*-MAS for Π, by Proposition 3.18, we have that $atom(l_a) \in A_{dpcy}$. Consider rule

$$r_\mathcal{M} \;=\; dpcy(l_a, l_a) \leftarrow atom(l_a)\,,$$

being a ground instance of rule (14). We have that $A_{dpcy} \models body(r_\mathcal{M})$. Since A_{dpcy} must satisfy $r_\mathcal{M}$, A_{dpcy} must contain the head of $r_\mathcal{M}$. Therefore, for each path within G of length 0 from atom a to itself we have that $dpcy(l_a, l_a) \in A_{dpcy}$.

Induction hypothesis: Assume for some arbitrary but fixed $n > 0$ that if there is a path from an atom a_0 to another atom a_n of length n in G, then $dpcy(l_{a_0}, l_{a_n}) \in A_{dpcy}$.

Step case: Consider a path $\Psi = a_0, a_1, \ldots, a_n, a_{n+1}$ of length $n+1$ in G. Since $a_0, a_1, \ldots, a_n$ is a path of length n in G, we conclude by induction hypothesis that $dpcy(l_{a_0}, l_{a_n}) \in A_{dpcy}$. By definition of a path, as a_n and a_{n+1} are two neighbouring elements in Ψ, $\langle a_n, a_{n+1}\rangle$ is an edge in G. Then, by definition of the positive dependency graph, there is some rule $r \in \Pi$ such that $a_n \in head(r)$ and $a_{n+1} \in body(r)$. Thus, since A_{dpcy} is an *aux*-MAS for Π, by Proposition 3.18, $head(l_r, l_{a_n}), bodyP(l_r, l_{a_{n+1}}) \in A_{dpcy}$. Consequently, for meta-rule

$$r_\mathcal{M} \;=\; dpcy(l_{a_n}, l_{a_{n+1}}) \leftarrow head(l_r, l_{a_n})\,, bodyP(l_r, l_{a_{n+1}})$$

of $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$, we have that $A_{dpcy} \models body(r_{\mathcal{M}})$. Since A_{dpcy} must satisfy $r_{\mathcal{M}}$, A_{dpcy} must contain the head of $r_{\mathcal{M}}$. Therefore, $dpcy(l_{a_n}, l_{a_{n+1}}) \in A_{dpcy}$.

Now consider meta-rule

$$r_{\mathcal{M}} \;=\; dpcy(l_{a_0}, l_{a_{n+1}}) \leftarrow dpcy(l_{a_0}, l_{a_n}), dpcy(l_{a_n}, l_{a_{n+1}})$$

of $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$. As $dpcy(l_0, l_{a_n}) \in A_{dpcy}$ and $dpcy(l_{a_n}, l_{a_{n+1}}) \in A_{dpcy}$, we have that $A_{dpcy} \models body(r_{\mathcal{M}})$. Since A_{dpcy} must satisfy $r_{\mathcal{M}}$, it must contain the head of $r_{\mathcal{M}}$. Therefore, $dpcy(l_{a_0}, l_{a_{n+1}}) \in A_{dpcy}$ holds. ■

Proposition 3.29 *Let A_{dpcy} be a dpcy-MAS for a propositional disjunctive program Π. Then, it holds that $dpcy(l_a, l_b) \in A_{dpcy}$ iff there is a path from a to b in the positive dependency graph of Π.*

Proof. Let $H_{\pi_{dpcy}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{dpcy}(\Pi))$, being a ground instance of a rule in π_{dpcy}. By Lemma 3.12, it holds that $A_{dpcy} \in AS((A_{dpcy} \setminus H_{\pi_{dpcy}}) \cup \pi_{dpcy})$. Furthermore, by Proposition 3.10, A_{dpcy} is an *aux*-MAS for Π. As A_{dpcy} is a *dpcy*-MAS, an *aux*-MAS for Π, and $\pi_{dpcy} \succ_{\mathcal{D}} \pi_{aux}$, by Lemma 3.11, we have that $A_{dpcy} \setminus H_{\pi_{dpcy}}$ is an *aux*-MAS for Π. From that, the fact that $\pi_{dpcy} \rhd (A_{dpcy} \setminus H_{\pi_{dpcy}})$, and since $A_{dpcy} \in AS((A_{dpcy} \setminus H_{\pi_{dpcy}}) \cup \pi_{dpcy})$, we conclude by Lemma 3.28 that $dpcy(l_a, l_b) \in A_{dpcy}$ iff there is a path from a to b in the positive dependency graph of Π. ■

Lemma 3.30 *Let $A_{\mathcal{M}}$ be an aux-MAS for a propositional disjunctive program Π such that $\pi_{dpcy} \rhd A_{\mathcal{M}}$ and A_{dpcy} the single answer set of $A_{\mathcal{M}} \cup \pi_{dpcy}$. Then, we have $strCon(l_a, l_b) \in A_{dpcy}$ iff a and b are vertices of the same strongly connected component in the positive dependency graph of Π.*

Proof. Let G be the positive dependency graph of Π.

($\Rightarrow$) Assume that $strCon(l_a, l_b) \in A_{dpcy}$ holds. There is only one rule $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$ which may support $\{strCon(l_a, l_b)\}$, namely

$$r_{\mathcal{M}} \;=\; strCon(l_a, l_b) \leftarrow dpcy(l_a, l_b), dpcy(l_b, l_a)\,.$$

Since we assumed $strCon(l_a, l_b) \in A_{dpcy}$, it must hold that $A_{dpcy} \models body(r_{\mathcal{M}})$ and therefore $dpcy(l_a, l_b), dpcy(l_b, l_a) \in A_{dpcy}$. By Lemma 3.28, there are paths from a to b and from b to a in G. That means a and b are vertices of the same strongly connected component within G.

($\Leftarrow$) Now we show that if a and b are vertices of the same strongly connected component within G then $strCon(l_a, l_b) \in A_{dpcy}$. Assume a and b are vertices of the same strongly connected component within G. Then, there are paths from a to b and from b to a in G. By Lemma 3.28, we have that $dpcy(l_a, l_b), dpcy(l_b, l_a) \in A_{dpcy}$. Therefore, it holds that $A_{dpcy} \models body(r_{\mathcal{M}})$ for rule

$$r_{\mathcal{M}} \;=\; strCon(l_a, l_b) \leftarrow dpcy(l_a, l_b), dpcy(l_b, l_a)\,,$$

being a ground instance of rule (17) in $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$. Since A_{dpcy} is an answer set of $Gr(A_{\mathcal{M}} \cup \pi_{dpcy})$, we have that $r_{\mathcal{M}}$ must be satisfied by A_{dpcy}. Hence, it holds that $strCon(l_a, l_b) \in A_{dpcy}$. ■

Proposition 3.31 *Let A_{dpcy} be a dpcy-MAS for a propositional disjunctive program Π. Then, it holds that $strCon(l_a, l_b) \in A_{dpcy}$ iff a and b are vertices of the same strongly connected component in the positive dependency graph of Π.*

Proof. Let $H_{\pi_{dpcy}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{dpcy}(\Pi))$, being a ground instance of a rule in π_{dpcy}. By Lemma 3.12, it holds that $A_{dpcy} \in AS((A_{dpcy} \setminus H_{\pi_{dpcy}}) \cup \pi_{dpcy})$. Furthermore, by Proposition 3.10, A_{dpcy} is an *aux*-MAS for Π. As A_{dpcy} is a *dpcy*-MAS, an *aux*-MAS for Π, and $\pi_{dpcy} \succ_{\mathcal{D}} \pi_{aux}$, by Lemma 3.11, we have that $A_{dpcy} \setminus H_{\pi_{dpcy}}$ is an *aux*-MAS for Π. From that, the fact that $\pi_{dpcy} \rhd (A_{dpcy} \setminus H_{\pi_{dpcy}})$, and since $A_{dpcy} \in AS((A_{dpcy} \setminus H_{\pi_{dpcy}}) \cup \pi_{dpcy})$, we conclude by Lemma 3.30 that $strCon(l_a, l_b) \in A_{dpcy}$ iff a and b are vertices of the same strongly connected component in the positive dependency graph of Π. ■

3.3.10 Guessing Loops

Module π_{loop} guesses loops within the strongly connected components in the positive dependency graph of program Π. Later we check whether these loops are externally supported by Π with respect to some interpretation I, by using module π_{ext}.

Since nontrivial loops of a program Π may intersect or be subsets of each other, we cannot detect all of them within a single meta-answer-set without introducing involved enumeration strategies. Thus, we allow multiple meta-answer-sets, each of which is used to inspect at most one loop of Π.

Definition 3.32 Module π_{loop} consists of the following rules:

$$loop(A) \leftarrow not\ \overline{loop}(A), atom(A), \tag{18}$$

$$\overline{loop}(A) \leftarrow not\ loop(A), atom(A), \tag{19}$$

$$\leftarrow loop(A1), loop(A2), not\ strCon(A1, A2). \tag{20}$$

□

Lemma 3.33 *Let $A_{\mathcal{M}}$ be a dpcy-MAS for a propositional disjunctive program Π such that $\pi_{loop} \rhd A_{\mathcal{M}}$ and $A_{loop} \in AS(A_{\mathcal{M}} \cup \pi_{loop})$. If $loop(l_a) \in A_{loop}$, then $\Gamma = \{x \mid loop(l_x) \in A_{loop}\}$ is a loop of Π.*

Proof. Let G be the positive dependency graph of Π. Assume that $loop(l_a) \in A_{loop}$ and consider the set

$$\Gamma = \{x \mid loop(l_x) \in A_{loop}\}.$$

At first, we will show that $\Gamma \subseteq HB(\Pi)$. Take an arbitrary $x \in \Gamma$. Thus, we have $loop(l_x) \in A_{loop}$. There is only one rule $r_{\mathcal{M}}$ in $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ which may support $\{loop(l_x)\}$, namely

$$r_{\mathcal{M}} \;=\; loop(l_x) \leftarrow not\ \overline{loop}(l_x), atom(l_x).$$

It must hold that $A_{loop} \models body(r_{\mathcal{M}})$ and hence $atom(l_x) \in A_{loop}$. Since A_{loop} is an *aux*-MAS for Π, we know by Proposition 3.18 that $x \in HB(\Pi)$. Therefore, we have $\Gamma \subseteq HB(\Pi)$.

Now we will show that each $c \in \Gamma$ belongs to the same strongly connected component of G. Take two arbitrary $x, y \in \Gamma$. Thus, we have $loop(l_x), loop(l_y) \in A_{loop}$. Now consider the following constraint $c_{\mathcal{M}}$ in $Gr(A_{\mathcal{M}} \cup \pi_{loop})$:

$$c_{\mathcal{M}} \;=\; \leftarrow loop(l_x), loop(l_y), not\ strCon(l_x, l_y).$$

Since $A_{loop} \in AS(Gr(A_{\mathcal{M}} \cup \pi_{loop}))$, it cannot hold that $A_{loop} \models body(c_{\mathcal{M}})$. We already know that $loop(l_x), loop(l_y) \in A_{loop}$ and hence $strCon(l_x, l_y) \in A_{loop}$. Since A_{loop} is a *dpcy*-MAS, we know by Proposition 3.31 that x and y are vertices of the same strongly connected component within G.

From the assumption that $loop(l_a) \in A_{loop}$, we know that $a \in \Gamma$. Hence, Γ is a non-empty subset of the set of all vertices within a strongly connected component of G. By Proposition 2.29, we have that Γ is a loop of Π. ■

Proposition 3.34 *Let $A_{\mathcal{M}}$ be a loop-MAS for a propositional disjunctive program Π. If $loop(l_a) \in A_{loop}$, then $\Gamma = \{x \mid loop(l_x) \in A_{loop}\}$ is a loop of Π.*

Proof. Let $H_{\pi_{loop}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{loop}(\Pi))$, being a ground instance of a rule in π_{loop}. By Lemma 3.12, it holds that $A_{loop} \in AS((A_{loop} \setminus H_{\pi_{loop}}) \cup \pi_{loop})$. Furthermore, by Proposition 3.10, A_{loop} is a *dpcy*-MAS for Π. As A_{loop} is a *loop*-MAS, an *dpcy*-MAS for Π, and $\pi_{loop} \succ_{\mathcal{D}} \pi_{dpcy}$, by Lemma 3.11, we have that $A_{loop} \setminus H_{\pi_{loop}}$ is a *dpcy*-MAS for Π. From that, the fact that $\pi_{loop} \rhd (A_{loop} \setminus H_{\pi_{loop}})$, and since $A_{loop} \in AS((A_{loop} \setminus H_{\pi_{loop}}) \cup \pi_{loop})$, we conclude by Lemma 3.33 that if $loop(l_a) \in A_{loop}$, then $\Gamma = \{x \mid loop(l_x) \in A_{loop}\}$ is a loop of Π. ■

For associating meta-answer-sets with individual loops for Π, we introduce the notion of meta-answer-sets *guessing loops*:

Definition 3.35 Let $A_{\mathcal{M}}$ be a meta-answer-set for a propositional disjunctive program Π such that $A_{\mathcal{M}}$ is a *loop*-MAS for Π. We say that $A_{\mathcal{M}}$ *guesses loop* Γ iff $\Gamma = \{x \mid loop(l_x) \in A_{\mathcal{M}}\}$ and $\Gamma \neq \emptyset$. □

Proposition 3.36 *Let Γ be a loop of Π and $A_{\mathcal{M}}$ a dpcy-MAS for Π such that $\pi_{loop} \rhd A_{\mathcal{M}}$. Then, there is some $A_{loop} \in AS(A_{\mathcal{M}} \cup \pi_{loop})$ such that $loop(l_a) \in A_{loop}$ iff $a \in \Gamma$.*

Proof. Let G be the positive dependency graph of Π.

Consider the set

$$A_{loop} = A_{\mathcal{M}} \cup \{loop(l_a) \mid a \in \Gamma\} \cup \{\overline{loop}(l_a) \mid a \in HB(\Pi) \setminus \Gamma\}.$$

We have that $loop(l_a) \in A_{loop}$ iff $a, b \in \Gamma$. We will show that $A_{loop} \in AS(A_{\mathcal{M}} \cup \pi_{loop})$.

Note that $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ is an absolutely tight program, since the atoms of the facts in $A_{\mathcal{M}}$ have no outgoing edges in G, and the ground instances of rules (18), (19), and (20) only have edges to atoms in $A_{\mathcal{M}}$ in G.

In order to show that $A_{loop} \in AS(A_{\mathcal{M}} \cup \pi_{loop})$, since $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ is an absolutely tight program, by Theorem 2.34, it suffices to verify that A_{loop} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ and A_{loop} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$.

We first show that A_{loop} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{loop})$. Since $A_{\mathcal{M}} \subseteq A_{loop}$, the facts of $A_{\mathcal{M}}$ are satisfied by A_{loop}. The remaining rules of $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ are ground instances of the rules (18), (19), and (20).

First, consider some rule

$$r_{\mathcal{M}} \;=\; loop(l_a) \leftarrow not\ \overline{loop}(l_a), atom(l_a),$$

being a ground instance of rule (18). Assume that $A_{loop} \models body(r_{\mathcal{M}})$ but also $loop(l_a) \notin A_{loop}$ hold. Thus, $\overline{loop}(l_a) \notin A_{loop}$ and $atom(l_a) \in A_{loop}$. From that, by definition of A_{loop}, we have

that $atom(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we have that $a \in HB(\Pi)$. Since $\Gamma \subseteq HB(\Pi)$, it must hold that either $a \in \Gamma$ or a $\in HB(\Pi) \setminus \Gamma$. However, then, by definition of A_{loop}, we get that either $loop(l_a) \in A_{loop}$ or $\overline{loop}(l_a) \in A_{loop}$, both contradictions to our assumptions. Thus, ground instances of rule (18) are satisfied by A_{loop}.

Now consider some rule

$$r_{\mathcal{M}} \;=\; \overline{loop}(l_a) \leftarrow not\ loop(l_a), atom(l_a)\,,$$

being a ground instance of rule (19). Assume that $A_{loop} \models body(r_{\mathcal{M}})$ but also $\overline{loop}(l_a) \notin A_{loop}$ hold. Thus, $loop(l_a) \notin A_{loop}$ and $atom(l_a) \in A_{loop}$. From that, by definition of A_{loop}, we have that $atom(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we have that $a \in HB(\Pi)$. Since $\Gamma \subseteq HB(\Pi)$, it must hold that either $a \in \Gamma$ or a $\in HB(\Pi) \setminus \Gamma$. However, then, by definition of A_{loop}, we get that either $loop(l_a) \in A_{loop}$ or $\overline{loop}(l_a) \in A_{loop}$, both contradictions to our assumptions. Thus, ground instances of rule (19) are satisfied by A_{loop}.

Finally, consider rule

$$c_{\mathcal{M}} \;=\; \leftarrow loop(l_a), loop(l_b), not\ strCon(l_a, l_b)\,,$$

being a ground instance of rule (20). Assume that $A_{loop} \models body(r_{\mathcal{M}})$ holds. Thus, it holds that $loop(l_a), loop(l_b) \in A_{loop}$ and $strCon(l_a, l_b) \notin A_{loop}$. By choice of A_{loop}, we have that $a, b \in \Gamma$ and thus a and b belong to the same strongly connected component of G. Since $A_{\mathcal{M}} \subseteq A_{loop}$, from $strCon(l_a, l_b) \notin A_{loop}$, we also have $strCon(l_a, l_b) \notin A_{\mathcal{M}}$. Therefore, since $A_{\mathcal{M}}$ is a *dpcy*-MAS for Π, we get by Proposition 3.31 that a and b are not vertices of the same strongly connected component within G, which causes a contradiction to previous results. Therefore, the assumption that $A_{loop} \models body(r_{\mathcal{M}})$ does not hold, and hence ground instances of rule (20) are satisfied by A_{loop}. This concludes the proof that $A_{loop} \models Gr(A_{\mathcal{M}} \cup \pi_{loop})$.

We now show that A_{loop} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$. Since $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ is non-disjunctive, by Proposition 2.31 it is sufficient to show that for every $a_{\mathcal{M}} \in A_{loop}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ with respect to A_{loop}. Clearly, for all atoms $a_{\mathcal{M}} \in A_{loop}$, we have either $a_{\mathcal{M}} \in A_{\mathcal{M}}$, $a_{\mathcal{M}} \in \{loop(l_a) \mid a \in \Gamma\}$, or $a_{\mathcal{M}} \in \{\overline{loop}(l_a) \mid a \in HB(\Pi) \setminus \Gamma\}$. In case $a_{\mathcal{M}} \in A_{\mathcal{M}}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ with respect to A_{loop}, because $a_{\mathcal{M}} \leftarrow$ is a fact in $Gr(A_{\mathcal{M}} \cup \pi_{loop})$.

If $a_{\mathcal{M}} \in \{loop(l_a) \mid a \in \Gamma\}$, consider the rule

$$r_{\mathcal{M}} \;=\; loop(l_a) \leftarrow not\ \overline{loop}(l_a), atom(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{loop})$, being a ground instances of rule (18), where $a_{\mathcal{M}} = loop(l_a)$. From $loop(l_a) \in A_{loop}$, we know by definition of A_{loop} that $a \in \Gamma$. Therefore, we have that $a \in HB(\Pi)$, and since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we get that $atom(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{loop}$, it holds that $atom(l_a) \in A_{loop}$. From $a \in \Gamma$, we know that $a \notin HB(\Pi) \setminus \Gamma$, and hence we can conclude by definition of A_{loop} that $\overline{loop}(l_a) \notin A_{loop}$. Hence, $A_{loop} \models body(r_{\mathcal{M}})$, and therefore, for all $a_{\mathcal{M}} \in \{loop(l_a) \mid a \in \Gamma\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$.

If $a_{\mathcal{M}} \in \{\overline{loop}(l_a) \mid a \in HB(\Pi) \setminus \Gamma\}$, consider the rule

$$r_{\mathcal{M}} \;=\; \overline{loop}(l_a) \leftarrow not\ loop(l_a), atom(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{loop})$, being a ground instance of rule (19), where $a_{\mathcal{M}} = \overline{loop}(l_a)$. From $\overline{loop}(l_a) \in A_{loop}$, we know by definition of A_{loop} that $a \in HB(\Pi) \setminus \Gamma$. Therefore, we have $a \in HB(\Pi)$,

and since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we get that $atom(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{loop}$, it holds that $atom(l_a) \in A_{loop}$. From $a \notin \Gamma$ we can conclude by definition of A_{loop} that $loop(l_a) \notin A_{loop}$. Hence, $A_{loop} \models body(r_{\mathcal{M}})$, and therefore, for all $a_{\mathcal{M}} \in \{\overline{loop}(l_a) \mid a \in HB(\Pi) \setminus \Gamma\}$, $\{a_{\mathcal{M}}\}$ is supported by $A_{\mathcal{M}} \cup \pi_{loop}$.

We showed that A_{loop} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$. Since A_{loop} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{loop})$, A_{loop} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{loop})$, and $Gr(A_{\mathcal{M}} \cup \pi_{loop})$ is an absolutely tight program, by Theorem 2.34, we have that $A_{loop} \in AS(A_{\mathcal{M}} \cup \pi_{loop})$. ■

3.3.11 Detecting Violated Constraints

Module π_{ic} is used to identify violated constraints in program Π under interpretation I.

Definition 3.37 Module π_{ic} consists of the following rules:

$$violated(R) \leftarrow ap(R), not\ hasHead(R), \tag{21}$$

$$hasHead(R) \leftarrow head(R, \_). \tag{22}$$

□

Lemma 3.38 *Let $A_{\mathcal{M}}$ be an ap-MAS for a propositional disjunctive program Π guessing interpretation I such that $\pi_{ic} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{ic}$ has a single answer set A_{ic}, and $violated(l_r) \in A_{ic}$ iff r is a constraint in Π which is violated under I.*

Proof. Since $A_{\mathcal{M}} \cup \pi_{ic}$ is normal and stratified, we know by Proposition 2.39 that $A_{\mathcal{M}} \cup \pi_{ic}$ has a unique answer set A_{ic}.

($\Rightarrow$) Assume that $violated(l_r) \in A_{ic}$ holds. There is only rule

$$r_{\mathcal{M}} \;=\; violated(l_r) \leftarrow ap(l_r), not\ hasHead(l_r).$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ic})$, which may support $\{violated(l_r)\}$. It must hold that $A_{ic} \models body(r_{\mathcal{M}})$, hence $ap(l_r) \in A_{ic}$, and $hasHead(l_r) \notin A_{ic}$. Since A_{ic} is an an *ap*-MAS for Π guessing I, from Proposition 3.26, we know that r is an applicable rule in Π. It remains to show that r is a constraint. Towards a contradiction, assume that r is not a constraint, i.e., $head(r) \neq \emptyset$. Now, consider atom $a \in HB(\Pi)$ such that $a \in head(r)$. From Proposition 3.18, we know that $head(l_r, l_a) \in A_{ic}$. Thus, it holds that $A_{ic} \models body(r_{\mathcal{M}})$ for rule

$$r_{\mathcal{M}} \;=\; hasHead(l_r) \leftarrow head(l_r, l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ic})$, being a ground instance of rule (22). It must hold that $hasHead(l_r) \in A_{ic}$, however we have already shown that $hasHead(l_r) \notin A_{ic}$. The assumption that r is not a constraint was false, hence r is a constraint in Π, violated under I.

($\Leftarrow$) Assume r is a constraint in Π, violated under I. Now we show that $hasHead(l_r) \notin A_{ic}$. Towards a contradiction, assume that $hasHead(l_r) \in A_{ic}$. Only some rule

$$r_{\mathcal{M}} \;=\; hasHead(l_r) \leftarrow head(l_r, l_a).$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ic})$, being a ground instance of rule (22), which may support $\{hasHead(l_r)\}$. Under the current assumptions it must hold that $A_{ic} \models body(r_{\mathcal{M}})$ and therefore $head(l_r, l_a) \in A_{ic}$ for some $head(l_r, l_a) \in HB(A_{\mathcal{M}} \cup \pi_{ic})$. Since A_{ic} is an an *aux*-MAS for Π, we get by

Proposition 3.18 that $a \in head(r)$. However this is impossible since r is a constraint, hence the assumption that $hasHead(l_r) \in A_{ic}$ was false. We have $hasHead(l_r) \notin A_{ic}$.

We know that r is applicable in Π under I because r is violated in Π under I. Since A_{ic} is an an ap-MAS for Π guessing I, from Proposition 3.26, we know that $ap(l_r) \in A_{ic}$. We showed that $A_{ic} \models body(r_{\mathcal{M}})$ for rule

$$r_{\mathcal{M}} \;=\; violated(l_r) \leftarrow ap(l_r), not\ hasHead(l_r)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ic})$, being a ground instance of rule (21), and hence $violated(l_r) \in A_{ic}$ holds. ■

Proposition 3.39 *Let A_{ic} be an ic-MAS for a propositional disjunctive program Π guessing interpretation I. Then, $violated(l_r) \in A_{ic}$ holds iff r is a constraint in Π which is violated under I.*

Proof. Let $H_{\pi_{ic}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{ic}(\Pi))$, being a ground instance of a rule in π_{ic}. By Lemma 3.12, it holds that $A_{ic} \in AS((A_{ic} \setminus H_{\pi_{ic}}) \cup \pi_{ic})$. Furthermore, by Proposition 3.10, A_{ic} is an ap-MAS for Π. As A_{ic} is an ic-MAS, an ap-MAS for Π, and $\pi_{ic} \succ_{\mathcal{D}} \pi_{ap}$, by Lemma 3.11, we have that $A_{ic} \setminus H_{\pi_{ic}}$ is an ap-MAS for Π. Note that $A_{ic} \setminus H_{\pi_{ic}}$ guesses I. From that, the fact that $\pi_{ic} \rhd (A_{ic} \setminus H_{\pi_{ic}})$, and since $A_{ic} \in AS((A_{ic} \setminus H_{\pi_{ic}}) \cup \pi_{ic})$, we conclude by Lemma 3.38 that $violated(l_r) \in A_{ic}$ holds iff r is a constraint in Π which is violated under I. ■

3.3.12 Detecting Unsupported Atoms

Module π_{supp} is used to identify atoms which are unsupported by program Π with respect to interpretation I.

Definition 3.40 Module π_{supp} consists of the following rules:

$$otherHeadInI(R, A) \leftarrow head(R, A), head(R, A2), int(A), int(A2), A \neq A2, \tag{23}$$

$$oHOfApRinI(A) \leftarrow ap(R), int(A), head(R, A), not\ otherHeadInI(R, A), \tag{24}$$

$$unsupported(A) \leftarrow int(A), not\ oHOfApRinI(A). \tag{25}$$

□

Lemma 3.41 *Let $A_{\mathcal{M}}$ be an ap-MAS for a propositional disjunctive program Π guessing interpretation I such that $\pi_{supp} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{supp}$ has a unique answer set A_{supp}, and $unsupported(l_a) \in A_{supp}$ iff $a \in I$ and a is unsupported by Π with respect to I.*

Proof. Since $A_{\mathcal{M}} \cup \pi_{supp}$ is normal and stratified, we know by Proposition 2.39 that $A_{\mathcal{M}} \cup \pi_{supp}$ has a unique answer set, A_{supp}. It remains to show that $unsupported(l_a) \in A_{supp}$ iff $a \in I$ and a is unsupported by Π with respect to I.

($\Rightarrow$) Assume that $unsupported(l_a) \in A_{supp}$ holds. Thus, there must be a rule in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$ being applicable under A_{supp} such that $unsupported(l_a)$ is in the head of this rule. Only rule

$$r_{\mathcal{M}} \;=\; unsupported(l_a) \leftarrow int(l_a), not\ oHOfApRinI(l_a),$$

being a ground instance of rule (25), can have this property. Since $A_{supp} \models body(r_{\mathcal{M}})$, we have $int(l_a) \in A_{supp}$ and $oHOfApRinI(l_a) \notin A_{supp}$. Since A_{supp} guesses I, from $int(l_a) \in A_{supp}$ we know that $a \in I$, and hence also $a \in HB(\Pi)$.

We will show that a is unsupported by Π with respect to I. Towards a contradiction, assume that a is not unsupported by Π with respect to I. Then, there is a rule $r \in \Pi$ such that both, $I \cap head(r) = \{a\}$, and $I \models body(r)$. Note that, since $I \cap head(r) = \{a\}$, we have $a \in head(r)$. Since r is applicable under I and A_{supp} is an ap-MAS for Π guessing I, by Proposition 3.26, it holds that $ap(l_r) \in A_{supp}$. Moreover, as $a \in head(r)$ and A_{supp} is an aux-MAS for Π, by Proposition 3.18, it holds that $head(l_r, l_a) \in A_{supp}$. Consider rule

$$r_{\mathcal{M}} \;=\; oHOfApRinI(l_a) \leftarrow ap(l_r), int(l_a), head(l_r, l_a), not\ otherHeadInI(l_r, l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$, which is a ground instance of rule (24). Since we already derived $oHOfApRinI(l_a) \notin A_{supp}$, we know that A_{supp} does not satisfy the body of $r_{\mathcal{M}}$. In order for that, as we have already shown $ap(l_r) \in A_{supp}$, $int(l_a) \in A_{supp}$, and $head(l_r, l_a) \in A_{supp}$, it must hold that also $otherHeadInI(l_r, l_a) \in A_{supp}$. Thus, there must be a rule in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$ being applicable under A_{supp} such that $otherHeadInI(l_r, l_a)$ is in the head of this rule. Only ground instances of rule (23) can have this property. Consider such a rule

$$r_{\mathcal{M}} \;=\; otherHeadInI(l_r, l_a) \leftarrow head(l_r, l_a), head(l_r, l_b), int(l_a), int(l_b), l_a \neq l_b\,.$$

Since $A_{supp} \models body(r_{\mathcal{M}})$, we have that $head(l_r, l_a), head(l_r, l_b), int(l_b) \in A_{supp}$ and $l_a \neq l_b$. Since A_{supp} is an int-MAS for Π guessing I, we have by Proposition 3.21 that $b \in I$, and by Proposition 3.18, since A_{supp} is an aux-MAS for Π guessing I, we have $b \in head(r)$. We conclude that $b \in I \cap head(r)$. From uniqueness of the labels in meta-alphabet $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$ and $l_a \neq l_b$, we know that a and b are different atoms in $HB(\Pi)$. Hence, we have a contradiction to $I \cap head(r) = \{a\}$. Thus, the assumption that a is not unsupported by Π with respect to I has been incorrect.

($\Leftarrow$) Assume $a \in I$ and a is unsupported by Π with respect to I. By Definition 2.30, there is no rule $r \in \Pi$ such that both, $I \cap head(r) = \{a\}$, and $I \models body(r)$. From $a \in I$ and since A_{supp} is an int-MAS for Π guessing I, we have by Proposition 3.21 that $int(l_a) \in A_{supp}$.

As a first step, we will show that $oHOfApRinI(l_a) \notin A_{supp}$. Towards a contradiction, assume that $oHOfApRinI(l_a) \in A_{supp}$. Thus, there must be a rule in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$ being applicable under A_{supp} such that $oHOfApRinI(l_a)$ is in the head of this rule. Only ground instances of rule (24) can satisfy this condition. Consider such a rule,

$$r_{\mathcal{M}} \;=\; oHOfApRinI(l_a) \leftarrow ap(l_r), int(l_a), head(l_r, l_a), not\ otherHeadInI(l_r, l_a)\,.$$

From $A_{supp} \models body(r_{\mathcal{M}})$, we get $ap(l_r), head(l_r, l_a) \in A_{supp}$, and $otherHeadInI(l_r, l_a) \notin A_{supp}$. Since A_{supp} is an ap-MAS for Π guessing I, we have by Proposition 3.26 that r is an applicable rule in Π under I. Moreover, since A_{supp} is an aux-MAS for Π guessing I, we have $a \in head(r)$. From that, we know that $a \in I \cap head(r)$. As it is necessary, that not both, $I \cap head(r) = \{a\}$ and $I \models body(r)$, hold, since $I \models body(r)$ holds, and $a \in I \cap head(r)$, there must be another atom $b \in I \cap head(r)$. Consider rule

$$r_{\mathcal{M}} \;=\; otherHeadInI(l_r, l_a) \leftarrow head(l_r, l_a), head(l_r, l_b), int(l_a), int(l_b), l_a \neq l_b$$

in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$, being a ground instance of rule (23). Since $b \in I$ and A_{supp} is an int-MAS for Π guessing I, we have by Proposition 3.21 that $int(l_b) \in A_{supp}$. Moreover, since

$b \in head(r)$ and A_{supp} is an *aux*-MAS for Π guessing I, we have by Proposition 3.18 that $head(l_r, l_b) \in A_{supp}$. Since we already showed that $head(l_r, l_a), head(l_r, l_b), int(l_a), int(l_b) \in A_{supp}$, and by uniqueness of the labels l_a and l_b, we have that $A_{supp} \models body(r_{\mathcal{M}})$. Therefore, we have that $otherHeadInI(l_r, l_a) \in A_{supp}$, which is a contradiction to our previous result, that $otherHeadInI(l_r, l_a) \notin A_{supp}$. The assumption that $oHOfApRinI(l_a) \in A_{supp}$ has been incorrect, hence $oHOfApRinI(l_a) \notin A_{supp}$.

Now, consider rule

$$r_{\mathcal{M}} \;=\; unsupported(l_a) \leftarrow int(l_a), not\ oHOfApRinI(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{supp})$, being a ground instance of rule (25). Since we have $int(l_a) \in A_{supp}$ and $oHOfApRinI(l_a) \notin A_{supp}$, it holds that $A_{supp} \models body(r_{\mathcal{M}})$. Thus, $unsupported(l_a)$ is contained in A_{supp}. ■

Proposition 3.42 *Let A_{supp} be a supp-MAS for a propositional disjunctive program Π guessing interpretation I. Then, $unsupported(l_a) \in A_{supp}$ holds iff $a \in I$ and a is unsupported by Π with respect to I.*

Proof. Let $H_{\pi_{supp}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{supp}(\Pi))$, being a ground instance of a rule in π_{supp}. By Lemma 3.12, it holds that $A_{supp} \in AS((A_{supp} \setminus H_{\pi_{supp}}) \cup \pi_{supp})$. Furthermore, by Proposition 3.10, A_{supp} is an *ap*-MAS for Π. As A_{supp} is a *supp*-MAS, an *ap*-MAS for Π, and $\pi_{supp} \succ_{\mathcal{D}} \pi_{ap}$, by Lemma 3.11, we have that $A_{supp} \setminus H_{\pi_{supp}}$ is an *ap*-MAS for Π. Note that $A_{supp} \setminus H_{\pi_{supp}}$ guesses I. From that, the fact that $\pi_{supp} \rhd (A_{supp} \setminus H_{\pi_{supp}})$, and since $A_{supp} \in AS((A_{supp} \setminus H_{\pi_{supp}}) \cup \pi_{supp})$, we conclude by Lemma 3.41 that $unsupported(l_a) \in A_{supp}$ holds iff $a \in I$ and a is unsupported by Π with respect to I. ■

3.3.13 Detecting Unsatisfied Rules

Module π_{sat} is used to identify rules in program Π which are not satisfied by interpretation I.

Definition 3.43 Module π_{sat} consists of the following rules:

$$anyHeadInI(R) \leftarrow head(R, A), int(A), \tag{26}$$

$$unsatisfied(R) \leftarrow ap(R), not\ anyHeadInI(R) \tag{27}$$

□

Lemma 3.44 *Let $A_{\mathcal{M}}$ be an ap-MAS for a propositional disjunctive program Π guessing interpretation I such that $\pi_{sat} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{sat}$ has a single answer set A_{sat}, and $unsatisfied(l_r) \in A_{sat}$ holds iff we have that $r \in \Pi$, $I \models body(r)$ but $head(r) \cap I = \emptyset$.*

Proof. Since $A_{\mathcal{M}} \cup \pi_{sat}$ is normal and stratified, we know by Proposition 2.39 that $A_{\mathcal{M}} \cup \pi_{sat}$ has a unique answer set A_{sat}.

($\Rightarrow$) Assume that $unsatisfied(l_r) \in A_{sat}$. There is only rule

$$r_{\mathcal{M}} \;=\; unsatisfied(l_r) \leftarrow ap(l_r), not\ anyHeadInI(l_r)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{sat})$, being a ground instance of rule (27), which may support $\{unsatisfied(l_r)\}$. It must hold that $A_{sat} \models body(r_{\mathcal{M}})$ and hence we have $ap(l_r) \in A_{sat}$ and $anyHeadInI(l_r) \notin A_{sat}$.

Since A_{sat} is an *ap*-MAS for program Π guessing interpretation I, from $ap(l_r) \in A_{sat}$, by Proposition 3.26, rule $r \in \Pi$ is applicable under I and hence $I \models body(r)$.

It remains to show that $head(r) \cap I = \emptyset$. Assume that $head(r) \cap I \neq \emptyset$ holds. Hence there is an atom $a \in HB(\Pi)$ such that $a \in I$ and $a \in head(r)$. Since A_{sat} is an *aux*-MAS for program Π guessing interpretation I, $int(l_a) \in A_{sat}$, and by Proposition 3.18, we have $head(l_r, l_a) \in A_{sat}$. Therefore, we have $A_{sat} \models body(r_{\mathcal{M}})$ for rule

$$r_{\mathcal{M}} \;=\; anyHeadInI(l_r) \leftarrow head(l_r, l_a), int(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{sat})$, being a ground instance of rule (26). The body of $r_{\mathcal{M}}$ is satisfied by A_{sat}, thus it must hold that $anyHeadInI(l_r) \in A_{sat}$, however we already derived $anyHeadInI(l_r) \in A_{sat}$. Therefore, the assumption that $head(r) \cap I \neq \emptyset$ was false.

($\Leftarrow$) Assume that $r \in \Pi$, $I \models body(r)$, and $head(r) \cap I = \emptyset$. We now show that $anyHeadInI(l_r) \notin A_{sat}$. Towards a contradiction, assume that $anyHeadInI(l_r) \in A_{sat}$. Only some rule

$$r_{\mathcal{M}} \;=\; anyHeadInI(l_r) \leftarrow head(l_r, l_a), int(l_a)\,.$$

in $Gr(A_{\mathcal{M}} \cup \pi_{sat})$, being a ground instance of rule (26), may support $\{anyHeadInI(l_r)\}$. Since $anyHeadInI(l_r) \in A_{sat}$, it must hold that $head(l_r, l_a) \in A_{sat}$ and $int(l_a) \in A_{sat}$ for some $head(l_r, l_a), int(l_a) \in HB(A_{\mathcal{M}} \cup \pi_{sat})$. Since A_{sat} is an *aux*-MAS for program Π guessing interpretation I, we have $a \in I$, and by Proposition 3.18, we have $a \in head(r)$. Thus, we get $a \in head(r) \cap I$, which is a contradiction to $head(r) \cap I = \emptyset$. Thus, $anyHeadInI(l_r) \notin A_{sat}$.

From $I \models body(r)$ we know that rule r is applicable in Π under I. Since A_{sat} is an *ap*-MAS for program Π guessing interpretation I, by Proposition 3.26, we know that $ap(l_r) \in A_{sat}$.

We showed that $A_{sat} \models body(r_{\mathcal{M}})$ for rule

$$r_{\mathcal{M}} \;=\; unsatisfied(l_r) \leftarrow ap(l_r), not\ anyHeadInI(l_r)\,.$$

in $Gr(A_{\mathcal{M}} \cup \pi_{sat})$. Thus, it must hold that $unsatisfied(l_r) \in A_{sat}$. ■

Proposition 3.45 *Let A_{sat} be a sat-MAS for a propositional disjunctive program Π guessing interpretation I. Then, $unsatisfied(l_r) \in A_{sat}$ holds iff we have that $r \in \Pi$, $I \models body(r)$ but $head(r) \cap I = \emptyset$.*

Proof. Let $H_{\pi_{sat}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{sat}(\Pi))$, being a ground instance of a rule in π_{sat}. By Lemma 3.12, it holds that $A_{sat} \in AS((A_{sat} \setminus H_{\pi_{sat}}) \cup \pi_{sat})$. Furthermore, by Proposition 3.10, A_{sat} is an *ap*-MAS for Π. As A_{sat} is a *sat*-MAS, an *ap*-MAS for Π, and $\pi_{sat} \succ_{\mathcal{D}} \pi_{ap}$, by Lemma 3.11, we have that $A_{sat} \setminus H_{\pi_{sat}}$ is an *ap*-MAS for Π. Note that $A_{sat} \setminus H_{\pi_{sat}}$ guesses I. From that, the fact that $\pi_{sat} \rhd (A_{sat} \setminus H_{\pi_{sat}})$, and since $A_{sat} \in AS((A_{sat} \setminus H_{\pi_{sat}}) \cup \pi_{sat})$, we conclude by Lemma 3.44 that $unsatisfied(l_r) \in A_{sat}$ holds iff we have that $r \in \Pi$, $I \models body(r)$ but $head(r) \cap I = \emptyset$. ■

3.3.14 Detecting Externally Unsupported Loops

Module π_{ext} is used to identify loops which are not externally supported by program Π with respect to interpretation I.

Definition 3.46 Module π_{ext} consists of the following rules:

$$extSupp \leftarrow head(R, A), ap(R), loop(A), not\ anyBInLoop(R), not\ hInINotLoop(R), \quad (28)$$

$$anyBInLoop(R) \leftarrow bodyP(R, A), loop(A), \quad (29)$$

$$hInINotLoop(R) \leftarrow head(R, A), int(A), not\ loop(A), \quad (30)$$

$$loopNotInI \leftarrow loop(A), not\ int(A), \quad (31)$$

$$selfCaused(A) \leftarrow loop(A), not\ loopNotInI, not\ extSupp. \quad (32)$$

□

Lemma 3.47 *Let $A_{\mathcal{M}}$ be an ap-MAS and a loop-MAS for a propositional disjunctive program Π guessing interpretation I, and $\pi_{ext} \rhd A_{\mathcal{M}}$. Then, $A_{\mathcal{M}} \cup \pi_{ext}$ has a unique answer set A_{ext}. If $A_{\mathcal{M}}$ guesses no loop of Π, then there is no atom over predicate selfCaused/1 in A_{ext}. Otherwise, if $A_{\mathcal{M}}$ guesses loop Γ of Π, then for all $a \in \Gamma$, $selfCaused(a) \in A_{ext}$ iff $\Gamma \subseteq I$ and Γ is not externally supported by Π with respect to I.*

Proof. Since $A_{\mathcal{M}} \cup \pi_{ext}$ is normal and stratified, we know by Proposition 2.39 that $A_{\mathcal{M}} \cup \pi_{ext}$ has a unique answer set.

Assume no loop of Π is guessed by $A_{\mathcal{M}}$. Then, also no loop of Π is guessed by A_{ext}, and therefore by Definition 3.35, there is no atom over predicate $loop/1$ in A_{ext}. Towards a contradiction, assume there is an atom $selfCaused(l_a)$ in A_{ext}. There is only rule

$$r_{\mathcal{M}} \;=\; selfCaused(l_a) \leftarrow loop(l_a), not\ loopNotInI, not\ extSupp$$

in $A_{\mathcal{M}} \cup \pi_{ext}$, being a ground instance of rule (32), which may support $\{selfCaused(l_a)\}$. Since there is no atom over predicate $loop/1$ in A_{ext}, we have $loop(l_a) \notin A_{ext}$, and thus $r_{\mathcal{M}}$ is not applicable under A_{ext}. The assumption that $selfCaused(l_a)$ in A_{ext} was false, and hence there is no atom with predicate $selfCaused/1$ in A_{ext}.

Now, suppose Γ is a loop of Π and $A_{\mathcal{M}}$ guesses Γ. Consider the set A_{ext}:

$$\begin{aligned} A_{ext} = A_{\mathcal{M}} \cup & \\ & \{extSupp \mid \Gamma \text{ is externally supported by } \Pi \text{ with respect to } I\} \cup \\ & \{anyBInLoop(l_r) \mid r \in \Pi \text{ and } \Gamma \cap body^+(r) \neq \emptyset\} \cup \\ & \{hInINotLoop(l_r) \mid r \in \Pi \text{ and } (head(r) \cap I) \setminus \Gamma \neq \emptyset\} \cup \\ & \{loopNotInI \mid \Gamma \not\subseteq I\} \cup \\ & \{selfCaused(l_a) \mid a \in \Gamma \text{ and } \Gamma \subseteq I \text{ and } \Gamma \text{ is not externally} \\ & \qquad \text{supported by } \Pi \text{ with respect to } I\}. \end{aligned}$$

Note that we have that for all $a \in \Gamma$, $selfCaused(a) \in A_{ext}$ iff $\Gamma \subseteq I$ and Γ is not externally supported by Π with respect to I. Now we will prove that A_{ext} is the unique answer set of $A_{\mathcal{M}} \cup \pi_{ext}$.

Note that $A_{\mathcal{M}} \cup \pi_{ext}$ is an absolutely tight program, since the atoms of the facts in $A_{\mathcal{M}}$ have no outgoing edges in the positive dependency graph $G_{\mathcal{M}}$ of $A_{\mathcal{M}} \cup \pi_{ext}$, and the rules in $A_{\mathcal{M}} \cup \pi_{ext}$, being ground instances of rules (28), (29), (30), (31) and (32) from π_{ext}, only have edges to atoms in $A_{\mathcal{M}}$ within $G_{\mathcal{M}}$.

In order to show that $A_{ext} \in AS(A_{\mathcal{M}} \cup \pi_{ext})$, since $A_{\mathcal{M}} \cup \pi_{ext}$ is an absolutely tight program, by Theorem 2.34, it suffices to verify that A_{ext} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{ext})$ and A_{ext} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

We now show that A_{ext} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{ext})$. Since $A_{\mathcal{M}} \subseteq A_{ext}$ the facts of $A_{\mathcal{M}}$ are satisfied by A_{ext}. The remaining rules of $Gr(A_{\mathcal{M}} \cup \pi_{ext})$ are the ground instances of rules (28), (29), (30), (31) and (32) from π_{ext}.

Consider an arbitrary ground instance $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ext})$ of rule (28):

$$r_{\mathcal{M}} \;=\; \mathit{extSupp} \leftarrow \mathit{head}(l_r, l_a), \mathit{ap}(l_r), \mathit{loop}(l_a), \mathit{not}\ \mathit{anyBInLoop}(l_r), \mathit{not}\ \mathit{hInINotLoop}(l_r).$$

Towards a contradiction, assume that $A_{ext} \models \mathit{body}(r_{\mathcal{M}})$ but also $\mathit{extSupp} \notin A_{ext}$ hold. From this we get that $\mathit{head}(l_r, l_a) \in A_{ext}$, $\mathit{ap}(l_r) \in A_{ext}$, $\mathit{loop}(l_a) \in A_{ext}$, $\mathit{anyBInLoop}(l_r) \notin A_{ext}$, and $\mathit{hInINotLoop}(l_r) \notin A_{ext}$. By definition of A_{ext}, we have that $\mathit{head}(l_r, l_a) \in A_{\mathcal{M}}$, $\mathit{ap}(l_r) \in A_{\mathcal{M}}$, and $\mathit{loop}(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ is an *ap*-MAS for Π, by Proposition 3.18, we have that $r \in \Pi$, $a \in HB(\Pi)$, and $a \in \mathit{head}(r)$. From Proposition 3.26 we know, that r is an applicable rule in Π under I, hence $I \models \mathit{body}(r)$. Since $A_{\mathcal{M}}$ guesses Γ and $\mathit{loop}(l_a) \in A_{\mathcal{M}}$ it holds that $a \in \Gamma$. Since $\mathit{anyBInLoop}(l_r) \notin A_{ext}$ and $r \in \Pi$, we get by definition of A_{ext} that $\Gamma \cap \mathit{body}^+(r) = \emptyset$. Since $\mathit{hInINotLoop}(l_r) \notin A_{ext}$ and $r \in \Pi$, we get by definition of A_{ext} that $(\mathit{head}(r) \cap I) \setminus \Gamma \neq \emptyset$.

Summarising, we have that there is a rule $r \in \Pi$ such that $\mathit{head}(r) \cap \Gamma \neq \emptyset$, $I \models \mathit{body}(r)$, $\mathit{body}^+(r) \cap \Gamma = \emptyset$, and $I \cap (\mathit{head}(r) \setminus \Gamma) = \emptyset$. From that we get by definition of external support that Γ is externally supported by Π with respect to I. Thus, by definition of A_{ext}, it holds that $\mathit{extSupp} \in A_{ext}$ which is a contradiction to the assumption that $\mathit{extSupp} \notin A_{ext}$. Therefore, ground instances of rule (28) are satisfied by A_{ext}.

Consider an arbitrary ground instance $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ext})$ of rule (29):

$$r_{\mathcal{M}} \;=\; \mathit{anyBInLoop}(l_r) \leftarrow \mathit{bodyP}(l_r, l_a), \mathit{loop}(l_a).$$

Towards a contradiction, assume that $A_{ext} \models \mathit{body}(r_{\mathcal{M}})$ but also $\mathit{anyBInLoop}(l_r) \notin A_{ext}$ hold. Thus, $\mathit{bodyP}(l_r, l_a), \mathit{loop}(l_a) \in A_{ext}$. By definition of A_{ext}, we have that $\mathit{bodyP}(l_r, l_a), \mathit{loop}(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we have that $r \in \Pi$, $a \in HB(\Pi)$, and $a \in \mathit{body}^+(r)$. Since $A_{\mathcal{M}}$ guesses Γ and $\mathit{loop}(l_a) \in A_{\mathcal{M}}$ it holds that $a \in \Gamma$. Therefore, we have that $r \in \Pi$ and $\Gamma \cap \mathit{body}^+(r) \neq \emptyset$. Then, by definition of A_{ext} we get that $\mathit{anyBInLoop}(l_r) \in A_{ext}$, which is a contradiction to the assumption that $\mathit{anyBInLoop}(l_r) \notin A_{ext}$. Thus, ground instances of rule (29) are satisfied by A_{ext}.

Consider an arbitrary ground instance $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ext})$ of rule (30):

$$r_{\mathcal{M}} \;=\; \mathit{hInINotLoop}(l_r) \leftarrow \mathit{head}(l_r, l_a), \mathit{int}(l_a), \mathit{not}\ \mathit{loop}(l_a).$$

Towards a contradiction, assume that $A_{ext} \models \mathit{body}(r_{\mathcal{M}})$ but also $\mathit{hInINotLoop}(l_r) \notin A_{ext}$ hold. Thus, $\mathit{head}(l_r, l_a), \mathit{int}(l_a) \in A_{ext}$ and $\mathit{loop}(l_a) \notin A_{ext}$. Therefore, by definition of A_{ext}, we have that $\mathit{head}(l_r, l_a), \mathit{int}(l_a) \in A_{\mathcal{M}}$ and $\mathit{loop}(l_a) \notin A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π guessing I, we get $a \in I$ and by Proposition 3.18, we have that $r \in \Pi$, $a \in HB(\Pi)$ and $a \in \mathit{head}(r)$. Since $A_{\mathcal{M}}$ guesses Γ, we know from $\mathit{loop}(l_a) \notin A_{ext}$ that $a \notin \Gamma$. Therefore, we have that $r \in \Pi$ and $(\mathit{head}(r) \cap I) \setminus \Gamma \neq \emptyset$. Then, by definition of A_{ext} we get that $\mathit{hInINotLoop}(l_r) \in A_{ext}$, which is a contradiction to the assumption that $\mathit{hInINotLoop}(l_r) \notin A_{ext}$. Thus, ground instances of rule (30) are satisfied by A_{ext}.

Consider an arbitrary ground instance $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ext})$ of rule (31):

$$r_{\mathcal{M}} \;=\; \mathit{loopNotInI} \leftarrow \mathit{loop}(l_a), \mathit{not}\ \mathit{int}(l_a).$$

Towards a contradiction, assume that $A_{ext} \models body(r_{\mathcal{M}})$ but also $loopNotInI \notin A_{ext}$ hold. Thus, $loop(l_a) \in A_{ext}$ and $int(l_a) \notin A_{ext}$. Therefore, by definition of A_{ext}, we have that $loop(l_a) \in A_{\mathcal{M}}$ and $int(l_a) \notin A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ guesses I and Γ, we get $a \in \Gamma$ and $a \notin I$. Therefore, we have that $\Gamma \not\subseteq I$. Then, by definition of A_{ext} we get that $loopNotInI \in A_{ext}$, which is a contradiction to the assumption that $loopNotInI \notin A_{ext}$. Thus, ground instances of rule (31) are satisfied by A_{ext}.

Consider an arbitrary ground instance $r_{\mathcal{M}} \in Gr(A_{\mathcal{M}} \cup \pi_{ext})$ of rule (32):

$$r_{\mathcal{M}} \;=\; selfCaused(l_a) \leftarrow loop(l_a), not\ loopNotInI, not\ extSupp\,.$$

Towards a contradiction, assume that $A_{ext} \models body(r_{\mathcal{M}})$ but also $selfCaused(l_a) \notin A_{ext}$ hold. Thus, it follows that $loop(l_a) \in A_{ext}$, $loopNotInI \notin A_{ext}$, and $extSupp \notin A_{ext}$. Therefore, by definition of A_{ext}, we have that $loop(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}}$ guesses Γ, we get that (i) $a \in \Gamma$. By definition of A_{ext}, since $loopNotInI \in A_{ext}$, we get that (ii) $\Gamma \subseteq I$. Also by definition of A_{ext}, since $extSupp \notin A_{ext}$ we know that (iii) Γ is not externally supported by Π with respect to I. From (i), (ii), and (iii), we know by definition of A_{ext}, that $selfCaused(l_a) \in A_{ext}$, which is a contradiction to the assumption that $selfCaused(l_a) \notin A_{ext}$. Therefore, ground instances of rule (32) are satisfied by A_{ext}.

We proved that $A_{ext} \models Gr(A_{\mathcal{M}} \cup \pi_{ext})$, now we will show that A_{ext} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$. For all atoms $a_{\mathcal{M}} \in A_{ext}$, we have either

$a_{\mathcal{M}} \in A_{\mathcal{M}}$,

$a_{\mathcal{M}} \in \{extSupp \mid \Gamma \text{ is externally supported by } \Pi \text{ with respect to } I\}$,

$a_{\mathcal{M}} \in \{anyBInLoop(l_r) \mid r \in \Pi \text{ and } \Gamma \cap body^+(r) \neq \emptyset\}$,

$a_{\mathcal{M}} \in \{hInINotLoop(l_r) \mid r \in \Pi \text{ and } (head(r) \cap I) \setminus \Gamma \neq \emptyset\}$,

$a_{\mathcal{M}} \in \{loopNotInI \mid \Gamma \not\subseteq I\}$, or

$a_{\mathcal{M}} \in \{selfCaused(l_a) \mid a \in \Gamma \text{ and } \Gamma \subseteq I \text{ and } \Gamma \text{ is not externally supported by } \Pi \text{ with respect to } I\}$.

In case $a_{\mathcal{M}} \in A_{\mathcal{M}}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, because $a_{\mathcal{M}} \leftarrow$ is a fact in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

Consider the case where $a_{\mathcal{M}} \in \{extSupp \mid \Gamma \text{ is externally supported by } \Pi \text{ with respect to } I\}$. Since then $extSupp \in A_{ext}$, we have by definition of A_{ext} that Γ is externally supported by Π with respect to I. By definition of external support there is a rule $r \in \Pi$ such that $head(r) \cap \Gamma \neq \emptyset$, $I \models body(r)$, $body^+(r) \cap \Gamma = \emptyset$, and $I \cap (head(r) \setminus \Gamma) = \emptyset$. Take atom $a \in HB(\Pi)$ such that $a \in head(r) \cap \Gamma$. Since $A_{\mathcal{M}}$ is an *ap*-MAS for Π guessing I, by Proposition 3.18, we have that $head(l_r, l_a) \in A_{\mathcal{M}}$. From $I \models body(r)$, we know that r is an applicable rule in Π under I, hence by Proposition 3.26, we get $ap(l_r) \in A_{\mathcal{M}}$. Furthermore, since $A_{\mathcal{M}}$ guesses Γ, from $a \in \Gamma$, we get $loop(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{ext}$, it also holds that $head(l_r, l_a) \in A_{ext}$, $ap(l_r) \in A_{ext}$, and $loop(l_a) \in A_{ext}$. From $body^+(r) \cap \Gamma = \emptyset$, we know by definition of A_{ext} that $anyBInLoop(l_r) \notin A_{ext}$. From $I \cap (head(r) \setminus \Gamma) = \emptyset$, we know by definition of A_{ext} that $hInINotLoop(l_r) \notin A_{ext}$.

We showed that for rule

$$r_{\mathcal{M}} \;=\; extSupp \leftarrow head(l_r, l_a), ap(l_r), loop(l_a), not\ anyBInLoop(l_r), not\ hInINotLoop(l_r)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, being a ground instance of rule (28), it holds that $A_{ext} \models body(r_{\mathcal{M}})$. Therefore, for all $a_{\mathcal{M}} \in \{extSupp \mid \Gamma \text{ is externally supported by } \Pi \text{ with respect to } I\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

Consider the case where $a_{\mathcal{M}} \in \{anyBInLoop(l_r) \mid r \in \Pi \text{ and } \Gamma \cap body^+(r) \neq \emptyset\}$. From $anyBInLoop(l_r) \in A_{ext}$, we know by definition of A_{ext} that for rule $r \in \Pi$, $\Gamma \cap body^+(r) \neq \emptyset$. Take atom $a \in HB(\Pi)$ such that $a \in \Gamma \cap body^+(r)$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π, by Proposition 3.18, we have that $bodyP(l_r, l_a) \in A_{\mathcal{M}}$. Furthermore, since $A_{\mathcal{M}}$ guesses Γ, from $a \in \Gamma$, we get $loop(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{ext}$, it also holds that $bodyP(l_r, l_a) \in A_{ext}$ and $loop(l_a) \in A_{ext}$. We showed that for rule

$$r_{\mathcal{M}} \;=\; anyBInLoop(l_r) \leftarrow bodyP(l_r, l_a), loop(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, being a ground instance of rule (29), it holds that $A_{ext} \models body(r_{\mathcal{M}})$. Therefore, for all $a_{\mathcal{M}} \in \{anyBInLoop(l_r) \mid r \in \Pi \text{ and } \Gamma \cap body^+(r) \neq \emptyset\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

Consider the case where $a_{\mathcal{M}} \in \{hInINotLoop(l_r) \mid r \in \Pi \text{ and } (head(r) \cap I) \setminus \Gamma \neq \emptyset\}$. From $hInINotLoop(l_r) \in A_{ext}$, we know by definition of A_{ext} that for rule $r \in \Pi$, $(head(r) \cap I) \setminus \Gamma \neq \emptyset$. Take atom $a \in HB(\Pi)$ such that $a \in (head(r) \cap I) \setminus \Gamma$. Since $A_{\mathcal{M}}$ is an *aux*-MAS for Π guessing I and $a \in I$, we get $int(l_a) \in A_{ext}$ and by Proposition 3.18, we have that $head(l_r, l_a) \in A_{\mathcal{M}}$. Furthermore, since $A_{\mathcal{M}}$ guesses Γ, from $a \notin \Gamma$, we get $loop(l_a) \notin A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{ext}$, it also holds that $head(l_r, l_a), int(l_a) \in A_{ext}$. Furthermore, by definition of A_{ext} and $loop(l_a) \notin A_{\mathcal{M}}$, we also have $loop(l_a) \notin A_{ext}$. We showed that for rule

$$r_{\mathcal{M}} \;=\; hInINotLoop(l_r) \leftarrow head(l_r, l_a), int(l_a), not\ loop(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, being a ground instance of rule (30), it holds that $A_{ext} \models body(r_{\mathcal{M}})$. Therefore, for all $a_{\mathcal{M}} \in \{hInINotLoop(l_r) \mid r \in \Pi \text{ and } (head(r) \cap I) \setminus \Gamma \neq \emptyset\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

Consider the case where $a_{\mathcal{M}} \in \{loopNotInI \mid \Gamma \not\subseteq I\}$. Since then $loopNotInI \in A_{ext}$, we have by definition of A_{ext} that $\Gamma \not\subseteq I$. Thus, there must be some atom $a \in HB(\Pi)$ such that $a \in \Gamma$ but $a \notin I$. Since $A_{\mathcal{M}}$ guesses Γ, from $a \in \Gamma$, it holds that $loop(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{ext}$, it also holds that $loop(l_a) \in A_{ext}$. Since $A_{\mathcal{M}}$ guesses I, from $a \notin I$ we conclude that $int(l_a) \notin A_{\mathcal{M}}$. Hence, by definition of A_{ext}, we also have $int(l_a) \notin A_{ext}$.

We showed that for rule

$$r_{\mathcal{M}} \;=\; loopNotInI \leftarrow loop(l_a), not\ int(l_a)$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, being a ground instance of rule (31), it holds that $A_{ext} \models body(r_{\mathcal{M}})$. Therefore, for all $a_{\mathcal{M}} \in \{loopNotInI \mid \Gamma \not\subseteq I\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

Consider the case where $a_{\mathcal{M}} \in \{selfCaused(l_a) \mid a \in \Gamma \text{ and } \Gamma \subseteq I \text{ and } \Gamma \subseteq I \text{ and } \Gamma \text{ is not externally supported by } \Pi \text{ with respect to } I\}$. Since then $selfCaused(l_a) \in A_{ext}$, we have by definition of A_{ext} that for atom $a \in HB(\Pi)$, $a \in \Gamma$, $\Gamma \subseteq I$ and Γ is not externally supported by Π with respect to I. Thus, by definition of A_{ext}, we have $extSupp \notin A_{ext}$. Also by definition of A_{ext}, we know from $\Gamma \subseteq I$ that $loopNotInI \notin A_{ext}$. Since $A_{\mathcal{M}}$ guesses Γ, from $a \in \Gamma$, it holds that $loop(l_a) \in A_{\mathcal{M}}$. Since $A_{\mathcal{M}} \subseteq A_{ext}$, it also holds that $int(l_a), loop(l_a) \in A_{ext}$. We showed that for rule

$$r_{\mathcal{M}} \;=\; selfCaused(l_a) \leftarrow loop(l_a), not\ loopNotInI, not\ extSupp$$

in $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, being a ground instance of rule (32), it holds that $A_{ext} \models body(r_{\mathcal{M}})$. Therefore, for all $a_{\mathcal{M}} \in \{selfCaused(l_a) \mid a \in \Gamma \text{ and } \Gamma \subseteq I \text{ and } \Gamma \subseteq I \text{ and } \Gamma \text{ is not externally supported by } \Pi \text{ with respect to } I\}$, $\{a_{\mathcal{M}}\}$ is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$.

We showed that A_{ext} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$. Since A_{ext} satisfies $Gr(A_{\mathcal{M}} \cup \pi_{ext})$, A_{ext} is supported by $Gr(A_{\mathcal{M}} \cup \pi_{ext})$ and $Gr(A_{\mathcal{M}} \cup \pi_{ext})$ is an absolutely tight program by Theorem 2.34, we have that $A_{ext} \in AS(A_{\mathcal{M}} \cup \pi_{ext})$. ■

Proposition 3.48 *Let A_{ext} be an ext-MAS for a propositional disjunctive program Π guessing interpretation I. If $A_{\mathcal{M}}$ guesses no loop of Π, then there is no atom over predicate selfCaused/1 in A_{ext}. Otherwise, if $A_{\mathcal{M}}$ guesses loop Γ of Π, then for all $a \in \Gamma$, selfCaused$(a) \in A_{ext}$ iff $\Gamma \subseteq I$ and Γ is not externally supported by Π with respect to I.*

Proof. Let $H_{\pi_{ext}}$ be the set of atoms which occur in the head of a rule in $Gr(\mathcal{D}_{ext}(\Pi))$, being a ground instance of a rule in π_{ext}. By Lemma 3.12, it holds that $A_{ext} \in AS((A_{ext} \setminus H_{\pi_{ext}}) \cup \pi_{ext})$. Furthermore, by Proposition 3.10, A_{ext} is an *ap*-MAS for Π, and a *loop*-MAS for Π. As A_{ext} is an *ext*-MAS, an *ap*-MAS for Π, and $\pi_{ext} \succ_{\mathcal{D}} \pi_{ap}$, by Lemma 3.11, we have that $A_{ext} \setminus H_{\pi_{ext}}$ is an *ap*-MAS for Π. Moreover, as A_{ext} is an *ext*-MAS, a *loop*-MAS for Π, and $\pi_{ext} \succ_{\mathcal{D}} \pi_{loop}$, by Lemma 3.11, we have that $A_{ext} \setminus H_{\pi_{ext}}$ is a *loop*-MAS for Π. Note that $A_{ext} \setminus H_{\pi_{ext}}$ guesses I. From that, the fact that $\pi_{ext} \rhd (A_{ext} \setminus H_{\pi_{ext}})$, and since $A_{ext} \in AS((A_{ext} \setminus H_{\pi_{ext}}) \cup \pi_{ext})$, we conclude by Lemma 3.47 that if $A_{\mathcal{M}}$ guesses no loop of Π, then there is no atom over predicate *selfCaused*/1 in A_{ext}. Otherwise, if $A_{\mathcal{M}}$ guesses loop Γ of Π, then for all $a \in \Gamma$, *selfCaused*$(a) \in A_{ext}$ iff $\Gamma \subseteq I$ and Γ is not externally supported by Π with respect to I. ■

3.3.15 All Together Now!

So far the discussed modules allow us to find different reasons why a particular interpretation I is no answer set of the program Π to debug. Now we will investigate the set of meta-answer-sets of the union

$$\mathcal{D}_{err}(\Pi) = \pi_{in(\Pi)} \cup \pi_{aux} \cup \pi_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}$$

of these modules.

Proposition 3.49 *Let Π be a propositional disjunctive logic program. Every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is an ic-MAS, a supp-MAS, a sat-MAS, and an ext-MAS for Π.*

Proof. Program $\mathcal{D}_{err}(\Pi)$ can be written as follows:

$$\mathcal{D}_{err}(\Pi) = \mathcal{D}_{ic}(\Pi) \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}.$$

It holds that $\pi_{dpcy} \cup \pi_{loop} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext} \rhd \mathcal{D}_{supp}(\Pi)$. Therefore, by Definition 3.9, every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is an *ic*-MAS for Π.

Program $\mathcal{D}_{err}(\Pi)$ can written as follows:

$$\mathcal{D}_{err}(\Pi) = \mathcal{D}_{supp}(\Pi) \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{sat} \cup \pi_{ext}.$$

It holds that $\pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{sat} \cup \pi_{ext} \rhd \mathcal{D}_{supp}(\Pi)$. Therefore, by Definition 3.9, every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is a *supp*-MAS for Π.

Program $\mathcal{D}_{err}(\Pi)$ can be written as follows:

$$\mathcal{D}_{err}(\Pi) = \mathcal{D}_{sat}(\Pi) \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{ext}.$$

It holds that $\pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{ext} \rhd \mathcal{D}_{sat}(\Pi)$. Therefore, by Definition 3.9, every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is a *sat*-MAS for Π.

Program $\mathcal{D}_{err}(\Pi)$ can be written as follows:

$$\mathcal{D}_{err}(\Pi) = \mathcal{D}_{ext}(\Pi) \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat}.$$

It holds that $\pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \rhd \mathcal{D}_{supp}(\Pi)$. Therefore, by Definition 3.9, every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is an *ext*-MAS for Π. ■

Proposition 3.50 *For every interpretation I for a propositional disjunctive program Π and every loop Γ of Π, there is an $A_{\mathcal{M}} \in AS(\mathcal{D}_{err}(\Pi))$ such that $A_{\mathcal{M}}$ is an ic-MAS, a supp-MAS, a sat-MAS, and an ext-MAS for Π guessing I and Γ.*

Proof. By Proposition 3.49 we know that every set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is an *ic*-MAS, a *supp*-MAS, a *sat*-MAS, and an *ext*-MAS. We now show that for every interpretation I for Π and every loop Γ of Π there is a set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ such that A_{err} guesses I and Γ.

Program $\mathcal{D}_{err}(\Pi)$ can be written as follows:

$$\mathcal{D}_{err}(\Pi) = \mathcal{D}_{int}(\Pi) \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}.$$

By Proposition 3.6 for potential usage, since $\pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext} \rhd \mathcal{D}_{int}(\Pi)$, we can express the set of answer sets $AS(\mathcal{D}_{err}(\Pi))$ as follows:

$$AS(\mathcal{D}_{err}(\Pi)) = \bigcup_{A_{int} \in AS(\mathcal{D}_{int}(\Pi))} AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}).$$

From Corollary 3.22, we know that for every interpretation I, there is an *int*-MAS for Π guessing I in $AS(\mathcal{D}_{int}(\Pi))$. Let I be an arbitrary interpretation for Π and $A_{int} \in AS(\mathcal{D}_{int}(\Pi))$ an *int*-MAS for Π guessing I. Note that we have for set of answer sets $S_{\text{int}} = AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext})$, that $S_{\text{int}} \subseteq AS(\mathcal{D}_{err}(\Pi))$. Furthermore, we have for every $A_{err} \in S_{\text{int}}$, that $A_{int} \subseteq A_{err}$, because A_{int} is a set of facts. Thus, since $\pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}$ does not involve any atom over the $int/1$-predicate, we know that no additional atom over the $int/1$-predicate can be contained in A_{err}. Therefore, A_{err} guesses I.

Since $(\pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}) \rhd (A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop})$, by Proposition 3.6, we can rewrite S_{int} as follows:

$$S_{\text{int}} = \bigcup_{A_{\mathcal{M}} \in AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop})} AS(A_{\mathcal{M}} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}). \tag{33}$$

Since $\pi_{loop} \rhd A_{int} \cup \pi_{ap} \cup \pi_{dpcy}$, we can rewrite $AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop})$ by Proposition 3.6 as follows:

$$AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop}) = \bigcup_{A_{\mathcal{M}} \in AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy})} AS(A_{\mathcal{M}} \cup \pi_{loop}). \tag{34}$$

Since $\pi_{dpcy} \rhd A_{int} \cup \pi_{ap}$, we can rewrite $AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy})$ by Proposition 3.6 as follows:

$$AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy}) = \bigcup_{A_{\mathcal{M}} \in AS(A_{int} \cup \pi_{ap})} AS(A_{\mathcal{M}} \cup \pi_{dpcy}). \tag{35}$$

By Lemma 3.25, since A_{int} is an *int*-MAS for Π guessing I, and $\pi_{ap} \rhd A_{int}$, we have that $A_{int} \cup \pi_{ap}$ has a unique answer set A_{ap} which is an *ap*-MAS for Π guessing I. We can rewrite equation (35) as follows:

$$AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy}) = AS(A_{ap} \cup \pi_{dpcy}).$$

By Lemma 3.28, since A_{ap} is an *aux*-MAS for Π and $\pi_{dpcy} \rhd A_{ap}$, we have that $A_{ap} \cup \pi_{dpcy}$ has a unique answer set A_{dpcy}, which is a *dpcy*-MAS for Π. Therefore, we have that

$$AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy}) = \{A_{dpcy}\}.$$

We can rewrite equation (34) as follows:

$$AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop}) = AS(A_{dpcy} \cup \pi_{loop}).$$

By Proposition 3.36, since A_{dpcy} is a *dpcy*-MAS for Π and $\pi_{loop} \rhd A_{dpcy}$, we have that there is an $A_{loop} \in AS(A_{dpcy} \cup \pi_{loop})$ such that A_{loop} is a *loop*-MAS for Π guessing Γ.

Remember equation (33):

$$S_{\text{int}} = \bigcup_{A_{\mathcal{M}} \in AS(A_{int} \cup \pi_{ap} \cup \pi_{dpcy} \cup \pi_{loop})} AS(A_{\mathcal{M}} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}).$$

We can deduce that for

$$S_{\text{loop}} = AS(A_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}),$$

we have $S_{\text{loop}} \subseteq S_{\text{int}}$. Therefore, every $A_{err} \in S_{\text{loop}}$ is an *int*-MAS for Π guessing I. Furthermore, since $S_{\text{int}} \subseteq AS(\mathcal{D}_{err}(\Pi))$, we have $S_{\text{loop}} \subseteq AS(\mathcal{D}_{err}(\Pi))$.

Finally, it remains to show that there is a set $A_{err} \in S_{\text{loop}}$ such that A_{err} is a *loop*-MAS for Π guessing Γ. Take an arbitrary answer set $A_{err} \in S_{\text{loop}}$. Since A_{loop} is a subprogram of $A_{loop} \cup \pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}$ consisting of facts only, we have $A_{loop} \subseteq A_{err}$. Since $\pi_{ic} \cup \pi_{supp} \cup \pi_{sat} \cup \pi_{ext}$ does not involve any atom over the *loop*/1-predicate, we know that no additional atom over the *loop*/1-predicate can be contained in A_{err}. Thus, since A_{loop} is a *loop*-MAS for Π guessing Γ, we also have $loop(l_a) \in A_{err}$ iff $loop(a) \in \Gamma$. From $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$, we know that A_{err} is a *supp*-MAS for Π, hence A_{err} is also a *loop*-MAS for Π. Therefore, by Definition 3.35, we have that A_{err} guesses Γ.

We showed that $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ is a *supp*-MAS, a *sat*-MAS, an *ic*-MAS, and an *ext*-MAS for Π guessing I and Γ. ■

Corollary 3.51 *For a propositional disjunctive program Π and an interpretation I for Π, $\mathcal{D}_{err}(\Pi)$ has one or more answer sets $A_{\mathcal{M}}$ such that $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I. Moreover, for every such $A_{\mathcal{M}}$ it holds that*

- *unsupported$(l_a) \in A_{\mathcal{M}}$ iff for atom $a \in I$, a is unsupported by Π with respect to I,*

- *unsatisfied*$(l_r) \in A_{\mathcal{M}}$ *iff for rule* $r \in \Pi$, $I \models body(r)$ *but* $head(r) \cap I = \emptyset$, *and*
- *violated*$(l_r) \in A_{\mathcal{M}}$ *iff* r *is a constraint in* Π, *violated under* I.

Corollary 3.52 *For a propositional disjunctive program* Π, *an interpretation* I *for* Π, *and a non-empty set* $\Gamma \in HB(\Pi)$ *of atoms, it holds that* Γ *is a loop of* Π *such that* Γ *is subset of* I *and* Γ *is not externally supported by* Π *with respect to* I *iff* $\mathcal{D}_{err}(\Pi)$ *has one or more answer sets* $A_{\mathcal{M}}$, *where* $A_{\mathcal{M}}$ *is a meta-answer-set for* Π *guessing* I *such that selfCaused*$(l_a) \in A_{\mathcal{M}}$ *exactly when* $a \in \Gamma$.

3.3.16 Filtering Out Non-Error-Indicating Meta-Answer-Sets

Module π_{noAS} is used to filter out meta-answer-sets which do not explain why the considered interpretation I is no answer set for program Π, regardless whether I really is (not) an answer set for Π. This is done because the absence of any atom over of the *error-indicating predicates* (*violated*/1, *unsupported*/1, *unsatisfied*/1, *selfCaused*/1) in a meta-answer-set $A_{\mathcal{M}} \in AS(\mathcal{D}_{err}(\Pi))$ for Π guessing I does not necessarily mean that I is no answer set for Π. More precisely since not all loops of Π are guessed within a single meta-answer-set, there may be a loop Γ of Π not detected by $A_{\mathcal{M}}$, which is not externally supported by Π with respect to I. By filtering out all meta-answer-sets including error-indicating-predicates, we get the nice property that for all interpretations I for Π there are *noAS*-MAS for Π guessing I in $AS(\mathcal{D}_{\mathcal{M}}(\Pi))$ iff I is not an answer set of Π.

Module π_{noAS} consists of the following rules:

$$noAnswerSet \leftarrow unsatisfied(\_), \tag{36}$$

$$noAnswerSet \leftarrow selfCaused(\_), \tag{37}$$

$$\leftarrow not\ noAnswerSet. \tag{38}$$

Lemma 3.53 *Let* Π *be a propositional disjunctive program. A set* $A_{\mathcal{M}}$ *is an answer set of* $\mathcal{D}_{\mathcal{M}}(\Pi)$ *iff* $A_{\mathcal{M}} = A_{err} \cup \{noAnswerSet\}$ *for an answer set* A_{err} *of* $\mathcal{D}_{err}(\Pi)$ *which contains at least one atom over one of the predicates unsatisfied*/1 *or selfCaused*/1.

Proof. ($\Rightarrow$) Let $A_{\mathcal{M}}$ be an answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$. Since $\mathcal{D}_{\mathcal{M}}(\Pi) = \mathcal{D}_{err}(\Pi) \cup \pi_{noAS}$ and $\pi_{noAS} \rhd \mathcal{D}_{err}(\Pi)$ we have by Proposition 3.6 that the following statement holds:

$$AS(\mathcal{D}_{\mathcal{M}}(\Pi)) = \bigcup_{A_{err} \in AS(\mathcal{D}_{err}(\Pi))} AS(A_{err} \cup \pi_{noAS}).$$

Therefore, there is some $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ such that $A_{\mathcal{M}} \in AS(A_{err} \cup \pi_{noAS})$.

Consider the following constraint $c_{\mathcal{M}}$ in π_{noAS}:

$$c_{\mathcal{M}} \ = \ \leftarrow not\ noAnswerSet.$$

Since $c_{\mathcal{M}}$ is satisfied by $A_{\mathcal{M}}$, it must hold that $noAnswerSet \in A_{\mathcal{M}}$. Only ground instances of rules (36) and (37) could possibly support $\{noAnswerSet\}$. Therefore, there must be a rule $r_{\mathcal{M}}$ in $Gr(A_{err} \cup \pi_{noAS})$, being a ground instance of rule (36) or (37) such that $A_{\mathcal{M}} \models body(r_{\mathcal{M}})$.

Assume there is a rule

$$r_{\mathcal{M}} \ = \ noAnswerSet \leftarrow unsatisfied(l_r)$$

in $Gr(A_{err} \cup \pi_{noAS})$, being a ground instance of rule (36) such that $A_{\mathcal{M}} \models body(r_{\mathcal{M}})$. Then, it must hold that $unsatisfied(l_r) \in A_{\mathcal{M}}$. Since $unsatisfied(l_r)$ is not contained in any ground instance of a rule in π_{noAS}, we have $unsatisfied(l_r) \in A_{err}$.

Assume there is a rule

$$r_{\mathcal{M}} \;=\; noAnswerSet \leftarrow selfCaused(l_a)$$

in $Gr(A_{err} \cup \pi_{noAS})$, being a ground instance of rule (37) such that $A_{\mathcal{M}} \models body(r_{\mathcal{M}})$. Then, it must hold that $selfCaused(l_a) \in A_{\mathcal{M}}$. Since $noAnswerSet$ is not contained in any ground instance of a rule in π_{noAS}, we have $selfCaused(l_a) \in A_{err}$. Therefore, there is either an atom $unsatisfied(l_r)$ or an atom $selfCaused(l_a)$ in A_{err}.

($\Leftarrow$) Let A_{err} be an answer set of $\mathcal{D}_{err}(\Pi)$ containing at least one atom over one of the predicates $unsatisfied/1$ or $selfCaused/1$. Consider the set

$$A_{\mathcal{M}} = A_{err} \;\cup \{noAnswerSet\}.$$

We have already shown the following equality:

$$AS(\mathcal{D}_{\mathcal{M}}(\Pi)) = \bigcup_{A_{err} \in AS(\mathcal{D}_{err}(\Pi))} AS(A_{err} \cup \pi_{noAS}).$$

From that, and since $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$, for showing that $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$, it suffices to show that $A_{\mathcal{M}} \in AS(A_{err} \cup \pi_{noAS})$. Note that $A_{err} \cup \pi_{noAS}$ is an absolutely tight program, since the atoms of the facts in A_{err} have no outgoing edges in the positive dependency graph G of $Gr(A_{err} \cup \pi_{noAS})$, and the ground instances of rules (36), (37) , and (38) from π_{noAS}, only have edges to atoms in A_{err} within G. In order to show that $A_{\mathcal{M}} \in AS(A_{err} \cup \pi_{noAS})$, since $Gr(A_{err} \cup \pi_{noAS})$ is an absolutely tight program, by Theorem 2.34, it suffices to verify that $A_{\mathcal{M}}$ satisfies $Gr(A_{err} \cup \pi_{noAS})$ and $A_{\mathcal{M}}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$. Since $A_{err} \subseteq A_{\mathcal{M}}$, all rules from A_{err} are satisfied by $A_{\mathcal{M}}$. We will now consider the remaining rules in $Gr(A_{err} \cup \pi_{noAS})$, the ground instances of rules (36), (37), and (38).

For arbitrary rule $r_{\mathcal{M}}$, being a ground instance of rule (36) or rule (37), we have that $noAnswerSet \in head(r_{\mathcal{M}})$. Since in any case $noAnswerSet \in A_{\mathcal{M}}$ by definition of $A_{\mathcal{M}}$, ground instances of rules (36) and (37) are satisfied by $A_{\mathcal{M}}$.

Consider the constraint

$$c_{\mathcal{M}} \;=\; \leftarrow not\ noAnswerSet$$

in $Gr(A_{err} \cup \pi_{noAS})$. Since in any case $noAnswerSet \in A_{\mathcal{M}}$ by definition of $A_{\mathcal{M}}$, it cannot hold that $A_{\mathcal{M}} \models body(c_{\mathcal{M}})$. Therefore, $c_{\mathcal{M}}$ cannot be violated under $A_{\mathcal{M}}$. Thus, $A_{\mathcal{M}}$ satisfies $Gr(A_{err} \cup \pi_{noAS})$.

Now we show that A_{int} is supported by $Gr(A_{aux} \cup \pi_{int})$. For all atoms $a_{\mathcal{M}} \in A_{\mathcal{M}}$, we have either $a_{\mathcal{M}} \in A_{err}$ or $a_{\mathcal{M}} = noAnswerSet$. In case $a_{\mathcal{M}} \in A_{err}$, $\{a_{\mathcal{M}}\}$ is trivially supported by $Gr(A_{err} \cup \pi_{noAS})$ because $a_{\mathcal{M}}$ is a fact in $Gr(A_{err} \cup \pi_{noAS})$.

Finally, we show that $\{noAnswerSet\}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$ with respect to $A_{\mathcal{M}}$.

Remember that we have either some $r \in \Pi$ such that $unsatisfied(l_r) \in A_{err}$ or some $a \in HB(\Pi)$ such that $selfCaused(l_a) \in A_{err}$. Therefore, we have by definition of $A_{\mathcal{M}}$ that there is either some atom $unsatisfied(l_r) \in A_{\mathcal{M}}$ or some atom $selfCaused(l_a) \in A_{\mathcal{M}}$.

Assume that $\mathit{unsatisfied}(l_r) \in A_{\mathcal{M}}$ holds and consider the following rule $r_{\mathcal{M}} \in Gr(A_{err} \cup \pi_{noAS})$:

$$r_{\mathcal{M}} \;=\; \mathit{noAnswerSet} \leftarrow \mathit{unsatisfied}(l_r).$$

Since $\mathit{head}(r_{\mathcal{M}}) = \{\mathit{noAnswerSet}\}$ and $A_{\mathcal{M}} \models \mathit{body}(r_{\mathcal{M}})$, $\{\mathit{noAnswerSet}\}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$.

Assume that $\mathit{selfCaused}(l_a) \in A_{\mathcal{M}}$ holds and consider the following rule $r_{\mathcal{M}} \in Gr(A_{err} \cup \pi_{noAS})$:

$$r_{\mathcal{M}} \;=\; \mathit{noAnswerSet} \leftarrow \mathit{selfCaused}(l_a).$$

Since $\mathit{head}(r_{\mathcal{M}}) = \{\mathit{noAnswerSet}\}$ and $A_{\mathcal{M}} \models \mathit{body}(r_{\mathcal{M}})$, $\{\mathit{noAnswerSet}\}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$. We showed that $A_{\mathcal{M}}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$. Since $A_{\mathcal{M}}$ satisfies $Gr(A_{err} \cup \pi_{noAS})$, $A_{\mathcal{M}}$ is supported by $Gr(A_{err} \cup \pi_{noAS})$ and $Gr(A_{err} \cup \pi_{noAS})$ is an absolutely tight program by Theorem 2.34, we have that $A_{\mathcal{M}} \in AS(A_{err} \cup \pi_{noAS})$. Thus, we have that $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$. ∎

Theorem 3.54 *Let I be an interpretation for a propositional disjunctive program Π. Then, I is no answer set of Π iff there is some noAS-MAS $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$ for program Π guessing interpretation I.*

Proof. ($\Rightarrow$) Assume I is no answer set of Π. By Theorem 2.33 either (i) I does not satisfy Π, or (ii) there is a loop $\Gamma \subseteq I$ of Π which is not externally supported by Π with respect to I. Consider case (i): we have that I does not satisfy Π. Then, there is some rule $r \in \Pi$ such that $r \in \Pi$, $I \models \mathit{body}(r)$ but $\mathit{head}(r) \cap I = \emptyset$. Hence by Corollary 3.51, there is a *sat*-MAS $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ for Π guessing I such that $\mathit{unsatisfied}(l_r) \in A_{err}$. Consider case (ii): we have that there is a loop $\Gamma \subseteq I$ of Π which is not externally supported by Π with respect to I. Hence by Corollary 3.52, there is an *ext*-MAS $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ for Π guessing I and Γ such that $\mathit{selfCaused}(l_a) \in A_{err}$ for all $a \in \Gamma$. Summarising cases (i) and (ii), we have that there is a meta-answer-set $A_{err} \in AS(\mathcal{D}_{err}(\Pi))$ for Π guessing I such that $\mathit{unsatisfied}(l_r) \in A_{err}$ for some $r \in \Pi$, or $\mathit{selfCaused}(l_a) \in A_{err}$ for some $a \in HB(\Pi)$. Consider the set

$$A_{\mathcal{M}} = A_{err} \;\cup \{\mathit{noAnswerSet}\}.$$

By Lemma 3.53 we have that $A_{\mathcal{M}}$ is an answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$. Since A_{err} is a meta-answer-set for Π guessing I, and $\{\mathit{noAnswerSet}\} \rhd A_{err}$, also $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I. Thus, we showed that $A_{\mathcal{M}}$ is a *noAS*-MAS for Π guessing I.

($\Leftarrow$) Assume $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$ is a *noAS*-MAS for program Π guessing I. Then, by Lemma 3.53, we have that $A_{\mathcal{M}} = A_{err} \cup \{\mathit{noAnswerSet}\}$ for an answer set A_{err} of $\mathcal{D}_{err}(\Pi)$ which contains at least one atom over one of the predicates $\mathit{unsatisfied}/1$ or $\mathit{selfCaused}/1$.

Assume that $\mathit{unsatisfied}(l_r) \in A_{err}$. By Proposition 3.49 we know that (i) A_{err} is a *sat*-MAS for Π, and thus by Proposition 3.10, A_{err} is also an *int*-MAS for Π. As $A_{\mathcal{M}}$ guesses I and since π_{noAS} does not contain atoms over predicate $\mathit{int}/1$, we have that (ii) $\mathit{int}(l_a) \in A_{err}$ iff $\mathit{int}(l_a) \in A_{\mathcal{M}}$. For (i) and (ii), by Definition 3.20, A_{err} is a *sat*-MAS guessing I. Thus, by Proposition 3.45, we have for rule $r \in \Pi$ that $I \models \mathit{body}(r)$ but $\mathit{head}(r) \cap I = \emptyset$. Hence $\{r\}$ is not satisfied by I and thus I is no answer set of Π.

Assume that $\mathit{selfCaused}(l_a) \in A_{err}$. By Proposition 3.49 we know that (i) A_{err} is an *ext*-MAS for Π, and thus by Proposition 3.10, A_{err} is also an *int*-MAS for Π. As $A_{\mathcal{M}}$ guesses I and since π_{noAS} does not contain atoms over predicate $\mathit{int}/1$, we have that (ii) $\mathit{int}(l_a) \in A_{err}$

iff $int(l_a) \in A_{\mathcal{M}}$. For (i) and (ii), by Definition 3.20, A_{err} is an *ext*-MAS guessing I. Thus, by Proposition 3.48, we have that there is a loop $\Gamma \subseteq I$ of Π such that Γ is not externally supported by Π with respect to I. Thus, by Theorem 2.33, I is no answer set of Π. ■

Each $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$ gives reasons, why an interpretation I is no answer set of program Π.

Theorem 3.55 *Let Π be a propositional disjunctive program. Every $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$ contains at least one atom over one of the two error-indicating predicates unsatisfied/1, selfCaused/1.*

Proof. Let $A_{\mathcal{M}}$ be an answer set of $AS(\mathcal{D}_{\mathcal{M}}(\Pi))$. Towards a contradiction, assume that for all meta-atoms $a_{\mathcal{M}} \in \{unsatisfied(l_r) \mid unsatisfied(l_r) \in HB(\mathcal{D}_{\mathcal{M}}(\Pi))\} \cup \{selfCaused(l_a) \mid selfCaused(l_a) \in HB(\mathcal{D}_{\mathcal{M}}(\Pi))\}$ over predicates *unsatisfied*/1 and *selfCaused*/1, it holds that $a_{\mathcal{M}} \notin A_{\mathcal{M}}$.

Assume that $noAnswerSet \in A_{\mathcal{M}}$ holds. Only ground instances of rules (36) and (37) may support $\{noAnswerSet\}$. Consider an arbitrary ground instance $r_{1\mathcal{M}} \in Gr(\mathcal{D}_{\mathcal{M}}(\Pi))$ of rule (36) and an arbitrary ground instance $r_{2\mathcal{M}} \in Gr(\mathcal{D}_{\mathcal{M}}(\Pi))$ of rule (37):

$$r_{1\mathcal{M}} = noAnswerSet \leftarrow unsatisfied(l_r),$$
$$r_{2\mathcal{M}} = noAnswerSet \leftarrow selfCaused(l_a).$$

We have $A_{\mathcal{M}} \not\models body(r_{1\mathcal{M}})$ and $A_{\mathcal{M}} \not\models body(r_{2\mathcal{M}})$. Therefore, $\{noAnswerSet\}$ is not supported by $Gr(\mathcal{D}_{\mathcal{M}}(\Pi))$ with respect to $A_{\mathcal{M}}$. Therefore, the assumption that $noAnswerSet \in A_{\mathcal{M}}$ was false, hence $noAnswerSet \in A_{\mathcal{M}}$.

Consider the following constraint $c_{\mathcal{M}} \in \mathcal{D}_{\mathcal{M}}(\Pi)$:

$$c_{\mathcal{M}} = \leftarrow not\ noAnswerSet.$$

It holds that $A_{\mathcal{M}} \models body(c_{\mathcal{M}})$, hence $A_{\mathcal{M}} \notin AS(\mathcal{D}_{\mathcal{M}}(\Pi))$, which is a contradiction to the assumption that $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$. Thus, for every $A_{\mathcal{M}} \in AS(\mathcal{D}_{\mathcal{M}}(\Pi))$, there is a meta-atom $a_{\mathcal{M}} \in \{unsatisfied(l_r) \mid unsatisfied(l_r) \in HB(\mathcal{D}_{\mathcal{M}}(\Pi))\} \cup \{selfCaused(l_a) \mid selfCaused(l_a) \in HB(\mathcal{D}_{\mathcal{M}}(\Pi))\}$ such that $a_{\mathcal{M}} \in A_{\mathcal{M}}$. ■

Theorem 3.56 *For a propositional disjunctive program Π and an interpretation I for Π such that I is no answer set of Π, $\mathcal{D}_{\mathcal{M}}(\Pi)$ has one or more answer sets $A_{\mathcal{M}}$ such that $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I. Moreover, for every such $A_{\mathcal{M}}$ it holds that*

- *$unsupported(l_a) \in A_{\mathcal{M}}$ iff for atom $a \in I$, a is unsupported by Π with respect to I,*
- *$unsatisfied(l_r) \in A_{\mathcal{M}}$ iff for rule $r \in \Pi$, $I \models body(r)$ but $head(r) \cap I = \emptyset$, and*
- *$violated(l_r) \in A_{\mathcal{M}}$ iff r is a constraint in Π, violated under I.*

Proof. By Theorem 3.54 it holds that $\mathcal{D}_{\mathcal{M}}(\Pi)$ has an answer set $A_{\mathcal{M}}$ such that $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I. As $A_{\mathcal{M}}$ is a *supp*-MAS, a *sat*-MAS, and an *ic*-MAS, by Propositions 3.42, 3.45, and 3.39, it holds that

- $unsupported(l_a) \in A_{\mathcal{M}}$ iff for atom $a \in I$, a is unsupported by Π with respect to I,
- $unsatisfied(l_r) \in A_{\mathcal{M}}$ iff for rule $r \in \Pi$, $I \models body(r)$ but $head(r) \cap I = \emptyset$, and

- *violated*$(l_r) \in A_{\mathcal{M}}$ iff r is a constraint in Π, violated under I. ■

Theorem 3.57 *For a propositional disjunctive program* Π*, an interpretation* I *for* Π*, and a non-empty set* $\Gamma \in HB(\Pi)$ *of atoms, it holds that* Γ *is a loop of* Π *such that* Γ *is subset of* I *and* Γ *is not externally supported by* Π *with respect to* I *iff* $\mathcal{D}_{\mathcal{M}}(\Pi)$ *has one or more answer sets* $A_{\mathcal{M}}$*, where* $A_{\mathcal{M}}$ *is a meta-answer-set for* Π *guessing* I *such that* $selfCaused(l_a) \in A_{\mathcal{M}}$ *for all* $a \in \Gamma$*.*

Proof. ($\Rightarrow$) Assume that $\Gamma \subseteq I$ and Γ is a loop of Π such that Γ is not externally supported by Π with respect to I. By Corollary 3.52, $\mathcal{D}_{err}(\Pi)$ has one or more answer sets A_{err}, where A_{err} is a meta-answer-set for Π guessing I such that $selfCaused(l_a) \in A_{err}$ for all $a \in \Gamma$. As Γ is non-empty, there is at least one atom over predicate $selfCaused/1$ in A_{err}. Therefore, by Lemma 3.53, $A_{\mathcal{M}} = A_{err} \cup \{noAnswerSet\}$ is an answer set of $\mathcal{D}_{\mathcal{M}}(\Pi)$. As A_{err} is a meta-answer-set for Π guessing I, so is $A_{\mathcal{M}}$. Therefore, $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I such that $selfCaused(l_a) \in A_{\mathcal{M}}$ for all $a \in \Gamma$.

($\Leftarrow$) Assume $\mathcal{D}_{\mathcal{M}}(\Pi)$ has an answer set $A_{\mathcal{M}}$, where $A_{\mathcal{M}}$ is a meta-answer-set for Π guessing I such that $selfCaused(l_a) \in A_{\mathcal{M}}$ for all $a \in \Gamma$. By Lemma 3.53, $A_{err} = A_{\mathcal{M}} \setminus \{noAnswerSet\}$ is an answer set of $\mathcal{D}_{err}(\Pi)$. As A_{err} is a meta-answer-set for Π guessing I such that $selfCaused(l_a) \in A_{\mathcal{M}}$ exactly when $a \in \Gamma$, we have by Corollary 3.52 that Γ is a loop of Π such that Γ is subset of I and Γ is not externally supported by Π with respect to I. ■

3.4 Search-Space Restriction and Examples

Since the user is confronted with a huge amount of information contained in $AS(\mathcal{D}_{\mathcal{M}}(\Pi))$, it makes sense to prune answer sets of $\mathcal{D}_{\mathcal{M}}(\Pi)$ which are not of interest to the user.

We distinguish between two approaches in restricting the amount of debugging information: Pruning of answer sets by constraints and looking only at optimal answer sets using weak constraints, respectively. In practice it is sometimes useful to apply both strategies simultaneously.

Generally, every meta-atom in $\mathcal{D}_{\mathcal{M}}(\Pi)$ provides a handle for directing the search for errors, allowing the user to specify the knowledge available about her or his expected results, e.g. which atoms are (not) contained in expected interpretations, and which rules are applicable, respectively blocked.

Example 3.58 Consider program Π_{ex}:

$$\begin{aligned}
\Pi_{ex} = \{\ r_1 &= a \leftarrow not\ b, c,\\
r_2 &= b \leftarrow not\ a,\\
r_3 &= c \leftarrow not\ d,\\
r_4 &= e \leftarrow c,\\
r_5 &= e \leftarrow d,\\
r_6 &= f \leftarrow c, not\ d,\\
r_7 &= g \leftarrow not\ h, e,\\
r_8 &= h \leftarrow not\ g,\\
r_9 &= \leftarrow a, g\ \}.
\end{aligned}$$

$\{int(l_b), int(l_c), int(l_f), int(l_h), noAnswerSet, unsatisfied(l_{r_4}),$
$ap(l_{r_2}), ap(l_{r_3}), ap(l_{r_4}), ap(l_{r_6}), ap(l_{r_8}),$
$bl(l_{r_1}), bl(l_{r_5}), bl(l_{r_7}), bl(l_{r_9}),$
$head(l_{r_1}, l_a), head(l_{r_2}, l_b), head(l_{r_3}, l_c), head(l_{r_4}, l_e),$
$head(l_{r_5}, l_e), head(l_{r_6}, l_f), head(l_{r_7}, l_g), head(l_{r_8}, l_h),$
$bodyP(l_{r_1}, l_c), bodyP(l_{r_4}, l_c), bodyP(l_{r_5}, l_d), bodyP(l_{r_6}, l_c),$
$bodyP(l_{r_7}, l_e), bodyP(l_{r_9}, l_a), bodyP(l_{r_9}, l_g),$
$bodyN(l_{r_1}, l_b), bodyN(l_{r_2}, l_a), bodyN(l_{r_3}, l_d),$
$bodyN(l_{r_6}, l_d), bodyN(l_{r_7}, l_h), bodyN(l_{r_8}, l_g),$
$atom(l_a), atom(l_b), atom(l_c), atom(l_d),$
$atom(l_e), atom(l_f), atom(l_g), atom(l_h),$
$rule(l_{r_1}), rule(l_{r_2}), rule(l_{r_3}), rule(l_{r_4}), rule(l_{r_5}),$
$rule(l_{r_6}), rule(l_{r_7}), rule(l_{r_8}), rule(l_{r_9}),$
$dpcy(l_a, l_a), dpcy(l_a, l_c), dpcy(l_b, l_b), dpcy(l_c, l_c), dpcy(l_d, l_d),$
$dpcy(l_e, l_c), dpcy(l_e, l_d), dpcy(l_e, l_e), dpcy(l_f, l_c), dpcy(l_f, l_f),$
$dpcy(l_g, l_c), dpcy(l_g, l_d), dpcy(l_g, l_e), dpcy(l_g, l_g), dpcy(l_h, l_h),$
$strCon(l_a, l_a), strCon(l_b, l_b), strCon(l_c, l_c), strCon(l_d, l_d),$
$strCon(l_e, l_e), strCon(l_f, l_f), strCon(l_g, l_g), strCon(l_h, l_h),$
$hasHead(l_{r_1}), hasHead(l_{r_2}), hasHead(l_{r_3}), hasHead(l_{r_4}),$
$hasHead(l_{r_5}), hasHead(l_{r_6}), hasHead(l_{r_7}), hasHead(l_{r_8}),$
$hInINotLoop(l_{r_2}), hInINotLoop(l_{r_3}), hInINotLoop(l_{r_6}), hInINotLoop(l_{r_8}),$
$anyHeadInI(l_{r_2}), anyHeadInI(l_{r_3}), anyHeadInI(l_{r_6}), anyHeadInI(l_{r_8}),$
$oHOfApRinI(l_b), oHOfApRinI(l_c), oHOfApRinI(l_f), oHOfApRinI(l_h),$
$\overline{loop}(l_a), \overline{loop}(l_b), \overline{loop}(l_c), \overline{loop}(l_d), \overline{loop}(l_e), \overline{loop}(l_f), \overline{loop}(l_g), \overline{loop}(l_h),$
$\overline{int}(l_a), \overline{int}(l_d), \overline{int}(l_e), \overline{int}(l_g)\}$

Figure 3.4: Answer set $A_{\Pi_{ex}}$ of program Π_{ex} from example 3.58

Assume the programmer wonders why $I = \{b, c, f, h\}$ is not an answer set of Π_{ex}. The corresponding meta-program $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ has 2277 answer sets. This number can be reduced to 9 by joining $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ with the following constraints, pruning all meta-answer-sets which are not guessing I:

$$\leftarrow int(a), \leftarrow int(d), \leftarrow int(e), \leftarrow int(g),$$
$$\leftarrow not\ int(b), \leftarrow not\ int(c), \leftarrow not\ int(f), \leftarrow not\ int(h).$$

The remaining meta-answer-sets only differ in which loop of Π they guess. As none of them includes an atom over the *selfCaused*/1-predicate, all loops of Π are externally supported with respect to I. Therefore, it is sufficient to focus on the remaining meta-answer-set guessing no loop, hence we add a further constraint, $\leftarrow loop(\_)$, and receive meta-answer-set $A_{\Pi_{ex}}$ which can be found in Figure 3.4. The only atom over an error-indicating predicate in $A_{\Pi_{ex}}$ is $unsatisfied(l_{r_4})$, stating that rule r_4 is not satisfied by I. ♢

Whenever the programmer is considering a specific interpretation I for program to debug Π, a good overview of the situation can be obtained by the following strategy. We join $\mathcal{D}_{\mathcal{M}}(\Pi)$ with the constraints $\leftarrow not\ int(l_a)$ for all $a \in I$, $\leftarrow int(l_b)$ for all $b \in HB(\Pi) \setminus I$, and the

following weak constraints:

$$:\sim atom(A), not\ selfCaused(A)\ [1:2], \tag{39}$$
$$:\sim loop(\_)\ [1:1]. \tag{40}$$

Weak constraint (39) expresses that, whenever there is a loop of Π which is a subset of I and not externally supported with respect to I, we obtain only meta-answer-sets indicating maximal loops of Π which are a subset of I and not externally supported with respect to I. If there is no such loop, weak constraint (40) becomes effective such that we obtain a single meta-answer-set guessing I, but not guessing any loop.

Example 3.59 Consider program

$$\begin{aligned}\Pi_{ex} = \{\ r_1 &= a \leftarrow b,\\ r_2 &= b \leftarrow c,\\ r_3 &= c \leftarrow a,\\ r_4 &= \leftarrow a, b\}\end{aligned}$$

and interpretation $I = \{a, b, c\}$. When we join $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ with the constraints $\leftarrow not\ int(a)$, $\leftarrow not\ int(b)$, $\leftarrow not\ int(c)$, the answer sets of the resulting program, projected to atoms over the predicates $int/1$, $unsupported/1$, $violated/1$, $unsatisfied/1$, $selfCaused/1$, and $loop/1$ are given by

$\{int(l_a), int(l_b), int(l_c), loop(l_a), loop(l_b), loop(l_c), selfCaused(l_a), selfCaused(l_b),$
$selfCaused(l_c), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_a), loop(l_b), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_a), loop(l_c), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_a), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_b), loop(l_c), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_b), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), loop(l_c), unsatisfied(l_{r4}), violated(l_{r4})\},$
$\{int(l_a), int(l_b), int(l_c), unsatisfied(l_{r4}), violated(l_{r4})\}.$

When we also add the weak constraints (39) and (40), only the set

$\{int(l_a), int(l_b), int(l_c), loop(l_a), loop(l_b), loop(l_c), selfCaused(l_a), selfCaused(l_b),$
$selfCaused(l_c), unsatisfied(l_{r4}), violated(l_{r4})\}$

remains, indicating that loop $\Gamma = \{a, b, c\}$ is not externally supported with respect to I. By adding the fact $\leftarrow a$ to Π_{ex}, which provides external support for Γ with respect to I, the resulting optimal answer set does not guess any loop. Its projection is given by

$$\{int(l_a), int(l_b), int(l_c), unsatisfied(l_{r4}), violated(l_{r4})\},$$

indicating the remaining error, namely the violation of constraint $r4$. ♢

In the next example, we do not restrict our attention to a single interpretation, but consider a specific class of interpretations.

Example 3.60 Reconsider program Π_{ex} from Section 3.1, consisting of the rules

$$\begin{aligned} r_1 &= night \vee day \leftarrow , \\ r_2 &= bright \leftarrow candlelight , \\ r_3 &= \leftarrow night, bright, not\ torch\_on , \\ r_4 &= candlelight \leftarrow . \end{aligned}$$

The programmer wants to know why there is no answer set A of Π_{ex} such that $night \in A$, constraint r_3 is not violated, and r_4 is applicable. This conditions can be expressed by the constraints

$$\begin{aligned} &\leftarrow not\ int(night), \\ &\leftarrow violated(l_{r_3}), \text{ and} \\ &\leftarrow not\ ap(l_{r_4}). \end{aligned}$$

As the union of $\mathcal{D}_{\mathcal{M}}(\Pi_{ex})$ and these constraints has 64 answer sets, we use the following weak constraints to consider only meta-answer-sets involving a minimal number of error-indicating meta-atoms:

$$:\sim violated(\_)\ [1:2], \tag{41}$$
$$:\sim unsupported(\_)\ [1:2], \tag{42}$$
$$:\sim unsatisfied(A), not\ violated(A)\ [1:2], \tag{43}$$
$$:\sim selfCaused(\_)\ [1:2], \tag{44}$$
$$:\sim loop(\_)\ [1:1]. \tag{45}$$

Weak constraints (41)-(44) minimise the number of error-indicating meta-atoms. Note, that weak constraint (43) only considers unsatisfied rules which are not constraints, as these are handled separately by weak constraint (39). Among the answer sets which are optimal with respect to weak constraints (41)-(44), weak constraint (45) gives preference to those involving a minimal number of atoms over predicate $loop/1$.

After applying these optimisations, only two optimal answer sets remain. These answer sets, projected to atoms over the predicates $int/1$, $unsupported/1$, $unsatisfied/1$, $violated/1$, $selfCaused/1$, $loop/1$, $ap/1$, and $bl/1$, are given by

$$\{ap(l_{r_1}), ap(l_{r_2}), ap(l_{r_4}), bl(l_{r_3}), int(l_{candlelight}), int(l_{night}), unsatisfied(l_{r_2})\} \quad \text{and}$$
$$\{ap(l_{r_1}), ap(l_{r_4}), bl(l_{r_2}), bl(l_{r_3}), int(l_{night}), unsatisfied(l_{r_4})\}. \qquad \diamondsuit$$

Chapter 4

Tagging Approach to Debugging

In previous work [5], we introduced a debugging method, based on a *tagging technique*, formerly used for compiling ordered logic programs into standard ones [15]. Similar to the meta-programming approach, described in Chapter 3, we use ASP techniques to debug propositional answer-set programs. In contrast to meta-programming, the tagging method does not fully lift the program Π to debug to the meta level, but relies on a variant of Π augmented with control and analysis atoms, called *tags*.

The general debugging question addressed by the basic version of the tagging technique concerns the applicability of rules with respect to the answer sets of Π: the debugging system states which rules are applicable and which rules are blocked for every answer set of Π. In the meta-programming approach, the same information can be obtained for all interpretations of Π, by computing the answer sets of standalone subprogram $\mathcal{D}_{ap}(\Pi)$ of the meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$.

The tagging technique can be extended to perform further sorts of debugging tasks. E.g., we propose an extension module for *extrapolation* of non-existing, yet putative answer sets of a program. The problem addressed here coincides with the major question motivating the meta-programming approach: "Why is some given interpretation not an answer set of a given program?". The approaches also use related means to tackle the problem: abnormalities regarding the program, its completion, and its loop formulas, as identified by the tagging technique, correspond to unsatisfied rules, unsupported atoms, and externally unsupported loops, detected in the meta-programming method. This scheme of justifications for an interpretation not to be an answer set is ultimately based on Theorem 2.37 (Lin-Zhao Theorem). However, in the meta-programming approach, we technically use a model-theoretic variant of this theorem by Lee [39]. Further details about the differences between the extrapolation module and the meta-programming technique are discussed in Section 4.2.

For inconsistent programs, the module for extrapolation of non-existing answer sets can be used to identify a minimal number of *repairs* to establish consistency. These are modifications of the program in form of either the deletion of a rule, or the addition of a fact. To this end, we use optimisation techniques of ASP-solvers.

In contrast to the meta-programming technique, where propositional disjunctive programs are handled, the class of programs to debug is restricted to propositional normal programs in the tagging approach. Integrity constraints must be handled by simulation.

For normal programs, the two approaches can usually be easily adapted to cover similar debugging queries. As the tagging technique is based on a rather slight translation of the

original program, where the candidate interpretations still depend on the applicability of rules, it is particularly suitable to analyse this coherence. E.g., following the work of Delgrande, Schaub, and Tompits [15], tagging can be used to impose an *order on the application of the rules* of a program. In terms of debugging, the question addressed here is which answer sets of Π can be thought of being computed sequentially, by a step-by-step application of rules, according to a rule order the developer has in mind. Although this could also be done by a meta-interpreter, the meta-technique we propose in Chapter 3 can not be adjusted for this purpose straightforwardly.

However generally, the clear separation of meta-level and program to debug makes the meta-approach more flexible and extensible than the tagging technique:

- once the program to debug is encoded, it does not have to be recompiled when switching debugging modules, and
- extension modules can be specified as abstract non-ground answer-set programs, without prior knowledge of the program to debug.

In the next section we introduce the basic version of our tagging approach.

4.1 Splitting Cause from Consequence

The main idea of tagging is to split the head from the body for each rule in a program, and thereby intervene in the applicability of rules. After this division, dedicated meta-atoms, called tags, are installed to trigger rules. This way, the formation of answer sets can be controlled and tags in the answer sets of the transformed (or *tagged*) program reflect inherent properties of Π.

Technically, a propositional normal program Π to debug over alphabet $\mathcal{A}$ is rewritten into a ground normal program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$. As the input language is restricted to normal programs but we also want to handle constraints, we need to simulate them. To this end, every constraint $\leftarrow \mathit{body}(c)$ is replaced by a rule $n \leftarrow \mathit{body}(c), \mathit{not}\ n$, where n is a globally new atom.

We call $\mathcal{T}_{\mathsf{K}}[\Pi]$ the *kernel tagging* or *kernel transformation* of Π, in order to distinguish it from further translations, extending $\mathcal{T}_{\mathsf{K}}[\Pi]$. It is defined as follows:

Definition 4.1 Let Π be a logic program over $\mathcal{A}$ and consider a bijection n, assigning each rule r over $\mathcal{A}$ a unique name n_r. Then, the program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for $r \in \Pi$, $b \in \mathit{body}^+(r)$, and $c \in \mathit{body}^-(r)$:

$$\mathit{head}(r) \leftarrow \mathsf{ap}(n_r), \mathit{not}\ \mathsf{ko}(n_r), \tag{46}$$

$$\mathsf{ap}(n_r) \leftarrow \mathsf{ok}(n_r), \mathit{body}(r), \tag{47}$$

$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), \mathit{not}\ b, \tag{48}$$

$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), \mathit{not}\ \mathit{not}\ c, \tag{49}$$

$$\mathsf{ok}(n_r) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_r). \tag{50}$$

□

The tags $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ intuitively express whether a rule r is applicable or blocked, respectively, while the *control tags* $\mathsf{ok}(n_r)$ and $\mathsf{ko}(n_r)$, also occurring in $\mathcal{T}_{\mathsf{K}}[\Pi]$, are used for manipulating the application of r.

Intuitively the rules of Π are split into rules of forms (46) and (47), separating the applicability of a rule from the actual occurrence of the respective rule head in an interpretation.

Analogously, rules of forms (48) and (49) elicit which rules are blocked. Tags, stating that rule r is applicable or blocked, are only derived if $\mathsf{ok}(n_r)$ holds, which is by default the case, as expressed by rules of form (50).

The results of the debugging process can be obtained from the answer sets of the tagged program. In particular the answer sets of $\mathcal{T}_\mathsf{K}[\Pi]$ extend the answer sets of Π by information about applicability of rules, as stated by the following theorem:

Theorem 4.2 ([5]) *Let Π be a logic program over $\mathcal{A}$. We have a one-to-one correspondence between the answer sets of Π and $\mathcal{T}_\mathsf{K}[\Pi]$ satisfying the following conditions:*

1. *If A is an answer set of Π, then*

$$A \cup \{\mathsf{ok}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \mathsf{GEN}(\Pi, A)\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \mathsf{GEN}(\Pi, A)\}$$

   is an answer set of $\mathcal{T}_\mathsf{K}[\Pi]$.

2. *If A_K is an answer set of $\mathcal{T}_\mathsf{K}[\Pi]$, then $A_\mathsf{K} \cap At(\mathcal{A})$ is an answer set of Π.*

Example 4.3 Consider the program

$$\begin{aligned}\Pi_\mathsf{K} = \{\ r_1 &= a \leftarrow b\,,\\ r_2 &= a \leftarrow not\ c\,,\\ r_3 &= b \leftarrow c, not\ d\,,\\ r_4 &= c \leftarrow not\ e\,,\\ r_5 &= e \leftarrow a\,,\\ r_6 &= d \leftarrow not\ b\ \},\end{aligned}$$

with the following set of answer sets:

$$AS(\Pi_\mathsf{K}) = \{\{a, d, e\}, \{c, d\}\}\ .$$

The kernel transformation of Π_K is listed in Figure 4.1. The answer sets of the tagged program are given by

$$\begin{aligned}AS(\mathcal{T}_\mathsf{K}[\Pi_\mathsf{K}]) = \{\{&a, d, e,\\ &\mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_5}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}),\\ &\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\},\\ \{&c, d,\\ &\mathsf{ap}(n_{r_4}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_5}),\\ &\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\}\}\ .\end{aligned}$$

From this results, the user knows by Theorem 4.2 that, e.g., rule r_2 of Π_K is applicable under $\{a, d, e\}$, but blocked under $\{c, d\}$. ♢

Extension modules can use the control tags $\mathsf{ok}(n_r)$ and $\mathsf{ko}(n_r)$ in $\mathcal{T}_\mathsf{K}[\Pi]$ to influence the formation of answer sets of the tagged program. This way, we can use tagging to address other

$a \leftarrow \mathsf{ap}(n_{r_1}), \mathit{not}\ \mathsf{ko}(n_{r_1})\,,$
$\mathsf{ap}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}), b\,,$
$\mathsf{bl}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}), \mathit{not}\ b\,,$
$\mathsf{ok}(n_{r_1}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_1})\,,$
$b \leftarrow \mathsf{ap}(n_{r_3}), \mathit{not}\ \mathsf{ko}(n_{r_3})\,,$
$\mathsf{ap}(n_{r_3}) \leftarrow \mathsf{ok}(n_{r_3}), c, \mathit{not}\ d\,,$
$\mathsf{bl}(n_{r_3}) \leftarrow \mathsf{ok}(n_{r_3}), \mathit{not}\ c\,,$
$\mathsf{bl}(n_{r_3}) \leftarrow \mathsf{ok}(n_{r_3}), \mathit{not}\ \mathit{not}\ d\,,$
$\mathsf{ok}(n_{r_3}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_3})\,,$
$e \leftarrow \mathsf{ap}(n_{r_5}), \mathit{not}\ \mathsf{ko}(n_{r_5})\,,$
$\mathsf{ap}(n_{r_5}) \leftarrow \mathsf{ok}(n_{r_5}), a\,,$
$\mathsf{bl}(n_{r_5}) \leftarrow \mathsf{ok}(n_{r_5}), \mathit{not}\ a\,,$
$\mathsf{ok}(n_{r_5}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_5})\,,$

$a \leftarrow \mathsf{ap}(n_{r_2}), \mathit{not}\ \mathsf{ko}(n_{r_2})\,,$
$\mathsf{ap}(n_{r_2}) \leftarrow \mathsf{ok}(n_{r_2}), \mathit{not}\ c\,,$
$\mathsf{bl}(n_{r_2}) \leftarrow \mathsf{ok}(n_{r_2}), \mathit{not}\ \mathit{not}\ c\,,$
$\mathsf{ok}(n_{r_2}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_2})\,,$
$c \leftarrow \mathsf{ap}(n_{r_4}), \mathit{not}\ \mathsf{ko}(n_{r_4})\,,$
$\mathsf{ap}(n_{r_4}) \leftarrow \mathsf{ok}(n_{r_4}), \mathit{not}\ e\,,$
$\mathsf{bl}(n_{r_4}) \leftarrow \mathsf{ok}(n_{r_4}), \mathit{not}\ \mathit{not}\ e\,,$
$\mathsf{ok}(n_{r_4}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_4})\,,$

$d \leftarrow \mathsf{ap}(n_{r_6}), \mathit{not}\ \mathsf{ko}(n_{r_6})\,,$
$\mathsf{ap}(n_{r_6}) \leftarrow \mathsf{ok}(n_{r_6}), \mathit{not}\ b\,,$
$\mathsf{bl}(n_{r_6}) \leftarrow \mathsf{ok}(n_{r_6}), \mathit{not}\ \mathit{not}\ b\,,$
$\mathsf{ok}(n_{r_6}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_6})$

Figure 4.1: Kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi_{\mathsf{K}}]$ of program Π_{K} from example 4.3

questions than whether a rule is applicable or blocked under a certain answer set. Generally, such extension modules can be seen as an encoding $\mathcal{D}[\Delta]$ of a debugging request Δ. From this point of view, we understand the tagging approach as a general framework for debugging purposes, where the debugging results are provided by the answer sets of the union of $\mathcal{T}_{\mathsf{K}}[\Pi]$ and $\mathcal{D}[\Delta]$. The workflow of this debugging model is shown in Figure 4.2.

In the following, we will discuss an extension module used to *extrapolate* non-existing answer sets of Π.

4.2 Extrapolation of Non-Existing Answer Sets

The idea behind this module is to break with the one-to-one correspondence between answer sets of the tagged program and the program Π to debug, and analyse why interpretations are *not* answer sets of Π. To identify reasons for that, we need categories of abnormality for interpretations, giving hints on the cause for them not being answer sets. Here, we rely on the characterisation of answer sets by Lin and Zhao [44], distinguishing between abnormalities resulting from the program, its completion, and its loop formulas. Technically, we introduce three *abnormality tags*, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$, corresponding to these three categories of error, which occur in the answer sets of the new translation $\mathcal{T}_{\mathsf{Ex}}[\Pi]$.

Let Π be a program over alphabet $\mathcal{A}$ and A_{Ex} an answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$, then

- $\mathsf{ab}_p(n_r) \in A_{\mathsf{Ex}}$ signals that rule r is unsatisfied by I,
- $\mathsf{ab}_c(a) \in A_{\mathsf{Ex}}$ points out that atom a is true but not supported with respect to I, and
- $\mathsf{ab}_l(a) \in A_{\mathsf{Ex}}$ aims at indicating an atom $a \subseteq I$ in the consequent of a violated loop formula of Π,

where $I = A_{\mathsf{Ex}} \cap At(\mathcal{A})$.

The overall module $\mathcal{T}_{\mathsf{Ex}}[\Pi]$ is a union of the kernel translation $\mathcal{T}_{\mathsf{K}}[\Pi]$, and three new translations, $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, of program Π to debug, defined as follows:

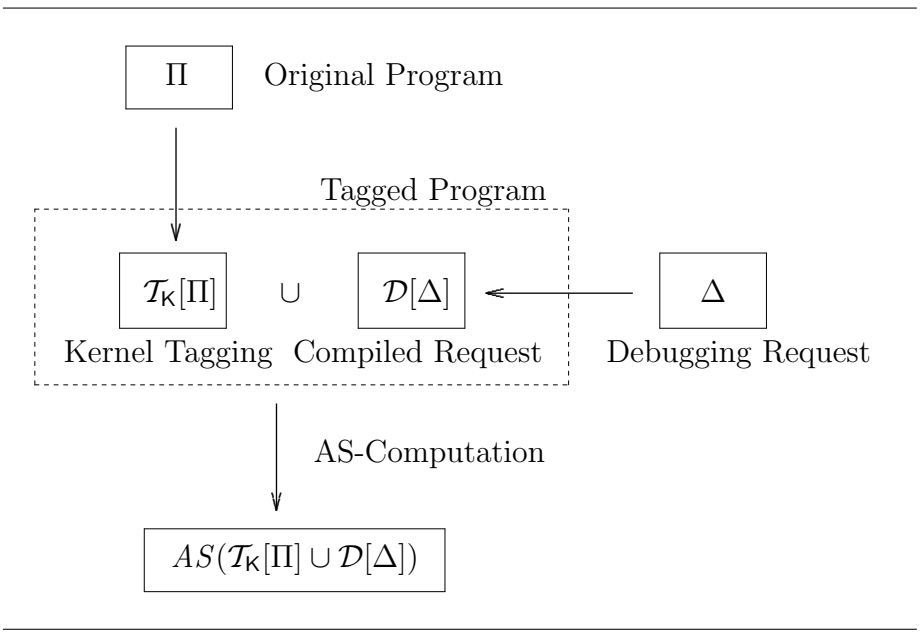

Figure 4.2: Debugging by tagging

Definition 4.4 Let Π be a logic program over $\mathcal{A}$. Then, for a set $A \subseteq At(\mathcal{A})$,

1. the logic program $\mathcal{T}_{\mathsf{P}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for all $r \in \Pi$

$$\{\, head(r) \,\} \leftarrow \mathsf{ap}(n_r)\,, \tag{51}$$
$$\mathsf{ab}_p(n_r) \leftarrow \mathsf{ap}(n_r), not\ head(r)\,, \tag{52}$$
$$\mathsf{ko}(n_r) \leftarrow\,, \tag{53}$$

2. the logic program $\mathcal{T}_{\mathsf{C}}[\Pi, A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$, where $\{r_1, \ldots, r_k\} = \{r \in \Pi \mid head(r) = \{a\}\}$:

$$\{\, a \,\} \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\,, \tag{54}$$
$$\mathsf{ab}_c(a) \leftarrow a, \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\,, \tag{55}$$

3. and the logic program $\mathcal{T}_{\mathsf{L}}[A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$:

$$\{\, \mathsf{ab}_l(a) \,\} \leftarrow not\ \mathsf{ab}_c(a)\,, \tag{56}$$
$$a \leftarrow \mathsf{ab}_l(a)\,. \tag{57}$$

□

The task of $\mathcal{T}_{\mathsf{P}}[\Pi]$ is to consider interpretations which do not satisfy all rules of the original program Π as answer sets. Technically, a rule of form (51) replaces a corresponding rule of form (46) from $\mathcal{T}_{\mathsf{K}}[\Pi]$. We use a choice rule to express that if rule r of Π is applicable the head of r may or may not be derived. In case it is not derived, the corresponding rule of

form (52) adds the abnormality tag $\mathsf{ab}_p(n_r)$ to the answer set. Rules of form (53) "knock out" the replaced rules of form (46).

$\mathcal{T}_\mathsf{C}[\Pi, A]$ is used to derive atoms which are not supported by any rule. If all rules having head atom $a \in A$ are blocked, a may or may not be derived by the corresponding choice rule of form (54). If a is derived, the abnormality tag $\mathsf{ab}_c(a)$ is true, due to the corresponding rule of form (55).

Finally, $\mathcal{T}_\mathsf{L}[A]$ is used to detect atoms in the consequent of violated loop formulas. Here, we randomly consider atoms to be abnormal and true, given that they are not already derived despite of being unsupported. Clearly, by rules of form (56), also atoms which are not involved in an unsatisfied loop formula are considered to be abnormal. However, when we consider only abnormality-minimum answer sets, the occurrence of an $\mathsf{ab}_l(a)$ tag always indicates the violation of a loop formula.

In the following, we only consider programs Π under an alphabet $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$. The set of all potential abnormality atoms for a propositional normal program Π under alphabet $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$ is given by

$$\mathsf{AB}(\Pi) = \{\mathsf{ab}_p(n_r) \mid r \in \Pi\} \cup \{\mathsf{ab}_c(a), \mathsf{ab}_l(a) \mid a \in HB(\Pi)\}.$$

We now define the extrapolation tagging of Π:

Definition 4.5 Let Π be a propositional normal program over $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$. Then, for sets $\Pi' \subseteq \Pi$ and $A \subseteq HB(\Pi)$, the *extrapolation tagging* of Π *projected to* Π' and A is given by

$$\mathcal{T}_\mathsf{Ex}[\Pi, \Pi', A] = \mathcal{T}_\mathsf{K}[\Pi] \cup \mathcal{T}_\mathsf{P}[\Pi'] \cup \mathcal{T}_\mathsf{C}[\Pi, A] \cup \mathcal{T}_\mathsf{L}[A] \ .$$

Moreover, the *extrapolation tagging* of Π is given by the program

$$\mathcal{T}_\mathsf{Ex}[\Pi] = \mathcal{T}_\mathsf{Ex}[\Pi, \Pi, HB(\Pi)]. \qquad \square$$

Using the extrapolation tagging projected to a subprogram and a set of atoms is useful to select the rules and atoms of a program which are estimated to behave incorrectly. Abnormality tags will only refer to the specified rules and atoms and the remaining program is assumed to be correct. The subsequent results deal with the special case of $\mathcal{T}_\mathsf{Ex}[\Pi]$, where all rules and atoms in Π are considered.

Theorem 4.6 ([5]) *Let Π be a logic program over $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$. Then, it holds that*

1. *If A is an answer set of Π, then*

   $$A \cup \{\mathsf{ok}(n_r), \mathsf{ko}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \mathsf{GEN}(\Pi, A)\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \mathsf{GEN}(\Pi, A)\}$$

   is an answer set of $\mathcal{T}_\mathsf{Ex}[\Pi]$.

2. *If A_Ex is an answer set of $\mathcal{T}_\mathsf{Ex}[\Pi]$ such that $A_\mathsf{Ex} \cap \mathsf{AB}(\Pi) = \emptyset$, then $A_\mathsf{Ex} \cap At(\mathcal{A})$ is an answer set of Π.*

Theorem 4.6 states that there is still a one-to-one correspondence between the answer sets of Π, and the answer sets of $\mathcal{T}_\mathsf{Ex}[\Pi]$ which do not involve abnormality tags. The next theorem deals with the case which is more interesting concerning the extrapolation behaviour of $\mathcal{T}_\mathsf{Ex}[\Pi]$, where abnormalities do occur.

Theorem 4.7 ([5]) *Let Π be a logic program over $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$. Then, it holds that*

1. *if A_{Ex} is an answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$ and $\mathsf{ab}_p(n_r) \in A_{\mathsf{Ex}}$, then $A_{\mathsf{Ex}} \cap At(\mathcal{A})$ is not a model of the formula $B_F(r) \rightarrow H_F(r)$ in $PF(\Pi)$;*
2. *if A_{Ex} is an answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$ and $\mathsf{ab}_c(a) \in A_{\mathsf{Ex}}$, then $A_{\mathsf{Ex}} \cap At(\mathcal{A})$ is not a model of $a \rightarrow \bigvee_{r \in \Pi, head(r)=\{a\}} B_F(r)$ in $CF(\Pi, \mathcal{A})$;*
3. *if A_{Ex} is an answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$ such that, for some $\Gamma \in loop_{\mathcal{NT}}(\Pi)$, we have $\Gamma \subseteq A_{\mathsf{Ex}} \cap At(\mathcal{A})$, $A_{\mathsf{Ex}} \cap At(\mathcal{A})$ is not a model of the loop formula $LF(\Pi, \Gamma)$, but $A_{\mathsf{Ex}} \cap At(\mathcal{A})$ is a model of $\bigvee_{r \in \Pi, head(r)=\{a\}} B_F(r)$ for every $a \in \Gamma$, then $\mathsf{ab}_l(a') \in A_{\mathsf{Ex}}$ for some $a' \in \Gamma$.*

While occurrences of abnormality tags $\mathsf{ab}_p(n_r)$ and $\mathsf{ab}_c(a)$ indicate a violation of $PF(\Pi)$ or $CF(\Pi, \mathcal{A})$, respectively, an occurrence of $\mathsf{ab}_l(a)$ does not necessarily indicate a violated loop formula. More precisely, having $\mathsf{ab}_l(a)$ in an answer set A_{Ex} of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$ means that a is also in $A_{\mathsf{Ex}} \cup At(\mathcal{A})$, a is supported by Π with respect to $A_{\mathsf{Ex}} \cup At(\mathcal{A})$, and either a is only supported by rules which recursively depend on a, or a is supported by a rule which does not depend on a. In the second case, a could also be derived, if $\mathsf{ab}_l(a)$ was not in A_{Ex}. As $\mathsf{ab}_l(a)$ was derived by a choice rule of form (56), $A_{\mathsf{Ex}} \setminus \{\mathsf{ab}_l(a)\}$ is also an answer set of the tagged program. Therefore, we can be sure that a loop formula was violated, when a tag $\mathsf{ab}_l(a)$ appears in an abnormality-minimum answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi]$.

In the meta-programming approach of Chapter 3 we do not encounter similar misleading "false positive" occurrences of error-indicating meta-atoms in the answer sets of $\mathcal{D}_{\mathcal{M}}(\Pi)$. This is because, unlike in tagging, there is an explicit check for loops being not externally supported. Furthermore, in tagging, atoms of distinct loops associated to violated loop formulas are detected, whereas in each answer set of the meta-program, one externally unsupported loop is found at most. Therefore, as discussed in Subsection 3.3.16, if we omitted module π_{noAS} we would obtain "false negative" results, namely meta-answer sets guessing an interpretation I which is not an answer set of Π, without detecting any error.

Another difference between the tagging and the meta-programming approach is that in the meta-approach unsupported atoms are regarded to be special cases of not externally supported loops, whereas, in terms of the corresponding notions, violations of $CF(\Pi, \mathcal{A})$ and violations of loop formulas are regarded to be distinct types of error.

Example 4.8 Consider the program

$$\begin{aligned} \Pi_{\mathsf{Ex1}} = \{\ r_1 &= a \leftarrow b, not\ c\,, \\ r_2 &= b \leftarrow a\,, \\ r_3 &= c \leftarrow not\ b\,, \\ r_4 &= b \leftarrow not\ a\ \}, \end{aligned}$$

which has no answer sets, and assume that the programmer does expect the existence of answer sets. The extrapolation tagging of Π_{Ex1} can be found in Figure 4.3. Furthermore, a complete list of the corresponding answer sets is given in Figure 4.4, including the following examples:

$$A_1 = \{c, \mathsf{ap}(n_{r_3}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{ab}_p(n_{r_4})\} \cup KOOK,$$

$$A_2 = \{b, c, \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_c(c)\} \cup KOOK,$$

$$A_3 = \{a, b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_l(a)\} \cup KOOK,$$

$$A_4 = \{b, c, \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_c(c), \mathsf{ab}_l(b)\} \cup KOOK,$$

$a \leftarrow \mathsf{ap}(n_{r_1}), \mathit{not}\ \mathsf{ko}(n_{r_1})$,
$\mathsf{ap}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}), b, \mathit{not}\ c$,
$\mathsf{bl}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}), \mathit{not}\ b$,
$\mathsf{bl}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}), \mathit{not\ not}\ c$,
$\mathsf{ok}(n_{r_1}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_1})$,
$c \leftarrow \mathsf{ap}(n_{r_3}), \mathit{not}\ \mathsf{ko}(n_{r_3})$,
$\mathsf{ap}(n_{r_3}) \leftarrow \mathsf{ok}(n_{r_3}), \mathit{not}\ b$,
$\mathsf{bl}(n_{r_3}) \leftarrow \mathsf{ok}(n_{r_3}), \mathit{not\ not}\ b$,
$\mathsf{ok}(n_{r_3}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_3})$,
$\{a\} \leftarrow \mathsf{ap}(n_{r_1})$,
$\mathsf{ab}_p(n_{r_1}) \leftarrow \mathsf{ap}(n_{r_1}), \mathit{not}\ a$,
$\{b\} \leftarrow \mathsf{ap}(n_{r_2})$,
$\mathsf{ab}_p(n_{r_2}) \leftarrow \mathsf{ap}(n_{r_2}), \mathit{not}\ b$,
$\{c\} \leftarrow \mathsf{ap}(n_{r_3})$,
$\mathsf{ab}_p(n_{r_3}) \leftarrow \mathsf{ap}(n_{r_3}), \mathit{not}\ c$,
$\{a\} \leftarrow \mathsf{bl}(n_{r_1})$,
$\mathsf{ab}_c(a) \leftarrow \mathsf{bl}(n_{r_1}), a$,
$\{b\} \leftarrow \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_4})$,
$\mathsf{ab}_c(b) \leftarrow \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_4}), b$,
$\{c\} \leftarrow \mathsf{bl}(n_{r_3})$,
$\mathsf{ab}_c(c) \leftarrow \mathsf{bl}(n_{r_3}), c$,

$b \leftarrow \mathsf{ap}(n_{r_2}), \mathit{not}\ \mathsf{ko}(n_{r_2})$,
$\mathsf{ap}(n_{r_2}) \leftarrow \mathsf{ok}(n_{r_2}), a$,
$\mathsf{bl}(n_{r_2}) \leftarrow \mathsf{ok}(n_{r_2}), \mathit{not}\ a$,
$\mathsf{ok}(n_{r_2}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_2})$,
$b \leftarrow \mathsf{ap}(n_{r_4}), \mathit{not}\ \mathsf{ko}(n_{r_4})$,
$\mathsf{ap}(n_{r_4}) \leftarrow \mathsf{ok}(n_{r_4}), \mathit{not}\ a$,
$\mathsf{bl}(n_{r_4}) \leftarrow \mathsf{ok}(n_{r_4}), \mathit{not\ not}\ a$,
$\mathsf{ok}(n_{r_4}) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_{r_4})$,

$\{b\} \leftarrow \mathsf{ap}(n_{r_4})$,
$\mathsf{ab}_p(n_{r_4}) \leftarrow \mathsf{ap}(n_{r_4}), \mathit{not}\ b$,
$\mathsf{ko}(n_{r_1}) \leftarrow$,
$\mathsf{ko}(n_{r_2}) \leftarrow$,
$\mathsf{ko}(n_{r_3}) \leftarrow$,
$\mathsf{ko}(n_{r_4}) \leftarrow$,
$\mathsf{ab}_l(a) \leftarrow \mathit{not}\ \mathsf{ab}_c(a)$,
$a \leftarrow \mathsf{ab}_l(a)$,
$\mathsf{ab}_l(b) \leftarrow \mathit{not}\ \mathsf{ab}_c(b)$,
$b \leftarrow \mathsf{ab}_l(b)$,
$\mathsf{ab}_l(c) \leftarrow \mathit{not}\ \mathsf{ab}_c(c)$,
$c \leftarrow \mathsf{ab}_l(c)$

Figure 4.3: Extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex1}}]$ of program Π_{Ex1} from example 4.8

where $KOOK = \{\mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2}), \mathsf{ko}(n_{r_3}), \mathsf{ko}(n_{r_4}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4})\}$.

Answer set A_1 represents interpretation $I_1 = \{c\}$ for Π. Abnormality tag $\mathsf{ab}_p(n_{r_4})$ indicates that rule r_4 is violated, as its head atom, b, is not in I_1. The tag $\mathsf{ab}_c(c)$ in answer set A_2 states that atom c is in the considered interpretation $I_2 = \{b, c\}$, although c does not occur in the head of a rule of Π which is applicable under I_2. Answer set A_3 represents interpretation $I_3 = \{a, b\}$ for Π. The occurrence of atom a in I_3 is due to the applicability of rule r_1 and hence dependent on the truth of b. Similarly, b is only true because a is, as r_2 is the only applicable rule with head b. Hence, a and b are in a loop associated to a violated loop formula in $LF(\Pi_{\mathsf{EX}})$ and therefore A_3 includes abnormality tag $\mathsf{ab}_l(a)$. Note that there is a dual answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{EX}}]$ which includes abnormality tag $\mathsf{ab}_l(b)$.

Furthermore, tag $\mathsf{ab}_l(b)$ in answer set A_4 is an example of a misleading loop-oriented abnormality tag, as b is not involved in a loop associated to a loop formula, violated under the represented interpretation $I_4 = \{b, c\}$. Thus, $A_4 \setminus \{\mathsf{ab}_l(b)\}$ is an answer set of $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex1}}]$, namely A_2. ♢

As stated before, in order to reduce the amount of information and to avoid misleading loop-oriented abnormality tags at the same time, we only consider abnormality-minimum answer sets of the extrapolation tagging. These can be computed, using standard optimisation techniques of ASP, such as *minimize statements* in `Smodels` or *weak constraints* in `DLV`.

The next result captures a scenario in which only a subset of a program is subject to extrapolation and only abnormality-minimum answer sets of the translation are considered. From the perspective of an original program Π, the abnormality-minimum answer sets of

$$\{a, b, c, \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_c(a), \mathsf{ab}_c(c)\} \cup KOOK,$$
$$\{a, b, c, \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_c(a), \mathsf{ab}_c(c), \mathsf{ab}_l(b)\} \cup KOOK,$$
$$\{a, b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_l(a)\} \cup KOOK,$$
$$\{a, b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_l(b)\} \cup KOOK,$$
$$\{a, b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_l(a), \mathsf{ab}_l(b)\} \cup KOOK,$$
$$\{a, c, \mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_c(a), \mathsf{ab}_p(n_{r_2})\} \cup KOOK,$$
$$\{a, c, \mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_c(a), \mathsf{ab}_l(c), \mathsf{ab}_p(n_{r_2})\} \cup KOOK,$$
$$\{a, \mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_4}), \mathsf{ab}_c(a), \mathsf{ab}_p(n_{r_2}), \mathsf{ab}_p(n_{r_3})\} \cup KOOK,$$
$$\{b, c, \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_c(c)\} \cup KOOK,$$
$$\{b, c, \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_c(c), \mathsf{ab}_l(b)\} \cup KOOK,$$
$$\{b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_p(n_{r_1})\} \cup KOOK,$$
$$\{b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{ab}_l(b), \mathsf{ab}_p(n_{r_1})\} \cup KOOK,$$
$$\{c, \mathsf{ap}(n_{r_3}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{ab}_p(n_{r_4})\} \cup KOOK,$$
$$\{c, \mathsf{ap}(n_{r_3}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{ab}_l(c), \mathsf{ab}_p(n_{r_4})\} \cup KOOK,$$
$$\{\mathsf{ap}(n_{r_3}), \mathsf{ap}(n_{r_4}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{ab}_p(n_{r_3}), \mathsf{ab}_p(n_{r_4})\} \cup KOOK, \quad \text{where}$$

$$KOOK = \{\mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2}), \mathsf{ko}(n_{r_3}), \mathsf{ko}(n_{r_4}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4})\}.$$

Figure 4.4: Answer sets of $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex1}}]$ of program Π_{Ex1} from example 4.8

$\mathcal{T}_{\mathsf{Ex}}[\Pi, \Pi', HB(\Pi')]$ provide us with the candidate sets among $At(\mathcal{A})$ that satisfy the requirement of being an answer set of Π under a minimum number of *repairs* on Π'. A repair is either the deletion of a rule r or an addition of a fact $a \leftarrow$ which prevents atom a from being not supported or in a loop associated to a violated loop formula. The former repair refers to an abnormality tag $\mathsf{ab}_p(n_r)$, which is clearly avoided when the corresponding rule r is deleted, and the latter to $\mathsf{ab}_c(a)$ or $\mathsf{ab}_l(a)$, since $a \rightarrow \bigvee_{r \in \Pi \cup \{a\leftarrow\}, head(r)=\{a\}} body(r)$ and $LF(\Pi \cup \{a \leftarrow\}, \Gamma)$, for any loop Γ containing a, then amount to $a \rightarrow \top$ and $\bot \rightarrow \bigwedge_{a \in \Gamma} \neg a$.

Theorem 4.9 ([5]) *Let Π be a logic program over $\mathcal{A}$ such that $HB(\Pi) = At(\mathcal{A})$ and (Π_1, Π_2) a partition of Π such that $\{head(r_1) \mid r_1 \in \Pi_1\} \cap HB(\Pi_2) = \emptyset$. Furthermore, let $\mathcal{M}$ be the set of all answer sets A_{Ex} of $\mathcal{T}_{\mathsf{Ex}}[\Pi, \Pi_2, HB(\Pi_2)]$ such that the cardinality of $A_{\mathsf{Ex}} \cap \mathsf{AB}(\Pi_2)$ is minimum among all answer sets of $\mathcal{T}_{\mathsf{Ex}}[\Pi, \Pi_2, HB(\Pi_2)]$. Then, it holds that*

1. *if $A_{\mathsf{Ex}} \in \mathcal{M}$, then $A_{\mathsf{Ex}} \cap At(\mathcal{A})$ satisfies all formulas in $PF(\Pi_1) \cup (CF(\Pi_1, \mathcal{A}) \setminus \{a \rightarrow \bot \mid a \in HB(\Pi_2)\}) \cup LF(\Pi_1)$ and all formulas in $PF(\Pi_2) \cup CF(\Pi_2, \mathcal{A}) \cup LF(\Pi_2)$ under a minimum number of repairs on Π_2;*
2. *if $I \subseteq At(\mathcal{A})$ satisfies all formulas in $PF(\Pi_1) \cup (CF(\Pi_1, \mathcal{A}) \setminus \{a \rightarrow \bot \mid a \in HB(\Pi_2)\}) \cup LF(\Pi_1)$ and all formulas in $PF(\Pi_2) \cup CF(\Pi_2, \mathcal{A}) \cup LF(\Pi_2)$ under a minimum number of repairs on Π_2, then there is a $A_{\mathsf{Ex}} \in \mathcal{M}$ such that $I = A_{\mathsf{Ex}} \cap At(\mathcal{A})$.*

This results are interesting in the context of inconsistent programs, as the number of minimal repairs for programs having answer sets is trivially zero.

Example 4.10 Consider the inconsistent program

$$\begin{aligned}\Pi_{\mathsf{Ex2}} = \{\ r_1 &= a \leftarrow \mathit{not}\ b\,,\\ r_2 &= b \leftarrow \mathit{not}\ c\,,\\ r_3 &= c \leftarrow \mathit{not}\ a\ \}.\end{aligned}$$

The extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex2}}, \{r_1\}, HB(\{r_1\})]$ of Π_{Ex2} projected to $\{r_1\}$ and $HB(\{r_1\})$ has two abnormality-minimal answer sets:

$$\begin{aligned}A_1{=}\{&b, c, \mathsf{ab}_c(b), \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{ko}(n_{r_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3})\} \quad \text{and}\\ A_2{=}\{&c, \mathsf{ab}_p(n_{r_1}), \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_2}), \mathsf{ko}(n_{r_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3})\}.\end{aligned}$$

The minimum number of repairs needed to restore consistency is one. From abnormality tag $\mathsf{ab}_c(b)$ in A_1 we know that $\Pi_{\mathsf{Ex2}} \cup \{b \leftarrow\}$ has the answer set $\{b, c\}$, and $\mathsf{ab}_p(n_{r_1})$ in A_2 indicates that $\Pi_{\mathsf{Ex2}} \setminus \{r_1\}$ has the answer set $\{c\}$. ♢

Chapter 5

Implementation

The approaches of Chapters 3 and 4 have been implemented in the prototype debugging system `spock` [6, 29]. The name `spock` makes reference to the fact that detecting errors is done by means of logic (just like the famous Vulcan of Star Trek always does), since the implemented techniques make use of ASP itself for debugging answer-set programs.

The system `spock` is a command-line oriented tool, written in *Java 5.0* and published under the *GNU general public license* [27]. It can be used either with `DLV` [40] or with `LPARSE`/`SMODELS` and is publicly available at

`http://www.kr.tuwien.ac.at/research/debug`

as a jar-package including binaries and sources.

5.1 Architecture

The main task of `spock` is performing translations between answer-set programs for debugging purposes. For a propositional disjunctive program Π_M, the meta-program $\mathcal{D}_{\mathcal{M}}(\Pi_M)$, defined in Chapter 3, can be computed. Furthermore, for propositional normal program Π_T and a subprogram Π'_T of Π_T, the translations $\mathcal{T}_{\mathsf{K}}[\Pi_T]$, $\mathcal{T}_{\mathsf{Ex}}[\Pi_T]$, $\mathcal{T}_{\mathsf{Ex}}[\Pi_T, \Pi'_T, HB(\Pi'_T)]$, $\mathcal{T}_{\mathsf{P}}[\Pi_T]$, $\mathcal{T}_{\mathsf{C}}[\Pi_T, HB(\Pi_T)]$ and $\mathcal{T}_{\mathsf{L}}[HB(\Pi_T)]$, as introduced in the tagging approach in Chapter 4, are supported. Furthermore, for the extrapolation tagging, `spock` allows for minimising the number of abnormality tags.

The data flow for all transformations is depicted by Figure 5.1. First, independent of the used translation, the input program is parsed and represented in an internal data structure. Then, the actual program transformation is performed, as specified by command-line arguments. The invocations of the individual translations are described later in this chapter.

The meta-programming approach as well as the tagging technique use labels to refer to individual rules. Therefore, we allow the programmer to add labels to the rules of the program to debug. As this requires an extension of the program syntax, `spock` offers an interface to `DLV` and `LPARSE`/`SMODELS` for computing answer sets of labelled programs. Figure 5.2 illustrates the typical data flow of answer-set computation with `spock`.

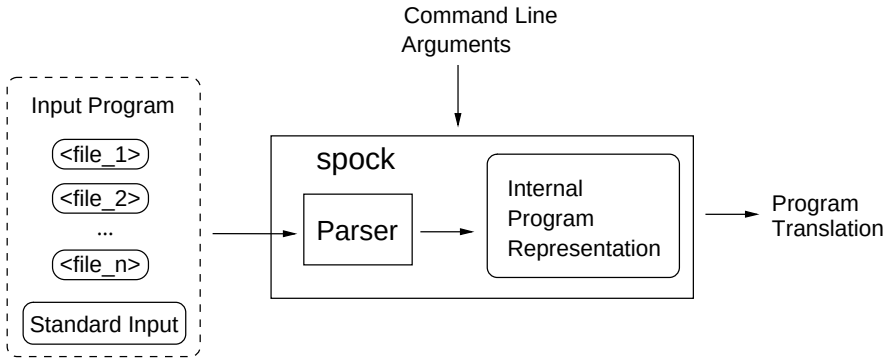

Figure 5.1: Data flow of program translations

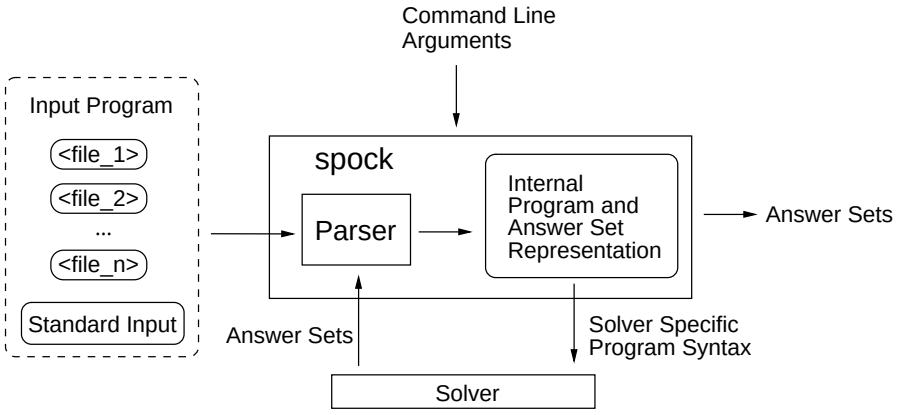

Figure 5.2: Data flow of answer-set computation for labelled programs

program	=	$(\ddots)^*$rule$((\ddots)^*rule)^*(\ddots)^*$
rule	=	(rulelabel ... ':' ...)? (head ... '.' |
		head ... ':-' ... body ... '.' | ':-' ... body ... '.')
head	=	atom (('v'|'|') ... atom)*
body	=	literal (',' ... literal)*
literal	=	atom | 'not' ... atom
atom	=	symb ('(' ... term(',' ... term)* ... ')')?
term	=	variable | symb
rulelabel	=	('a' – 'z' | 'A' – 'Z' | '0' – '9')*
variable	=	('A' – 'Z')('a' – 'z' | 'A' – 'Z' | '0' – '9' | '_')*
symb	=	('a' – 'z' | '0' – '9')('a' – 'z' | 'A' – 'Z' | '0' – '9' | '_')*
$\ddots$	=	(...)* '\n' (...)*
...	=	('␣' | '\t')*

Figure 5.3: Labelled program syntax of `spock`

5.2 System Call

Generally, `spock` is executed by a shell command of the form

```
java -jar spock.jar { OPTION | FILENAME }*,
```

where { OPTION | FILENAME }* is an arbitrary sequence of options and filenames, provided `java` is the execution command for the Java virtual machine. If no filename is given, `spock` expects input from the operation system's standard input. A list of important options is given in the Appendix.

5.3 System Input

The input primarily consists of the logic programs that are to be debugged. The accepted program syntax is closely related to the core languages of either `DLV` or `SMODELS`. Note that although `spock` allows a range of syntax constructs, like rule labels, head disjunctions, singular choice rules and weak constraints, the implemented debugging approaches work only on subsets of the accepted input.

The subset of the accepted program syntax which is needed for the implemented approaches, is depicted in Figure 5.3, using regular expressions.

Rule labelling is introduced as a device to explicitly refer to certain rules. As stated in Figure 5.3, a rule may have its label omitted. For a previously unlabelled rule, `spock` automatically assigns the label `r`n according to the line number n in which it appears in the program. Note that duplicate rule labels will produce a warning message. If the input is spread over multiple input files, their contents will be internally joined as if it was only one file. Additional content read from standard input when using the '`--`' flag is also appended to any input from files.

Since labelled rules cannot be read by conventional ASP solvers, `spock` offers an interface to `DLV` and `SMODELS` providing answer-set computation for labelled programs, described next.

5.4 Answer-Set Computation for Labelled Programs

In order to perform answer-set computation for labelled programs, either DLV or SMODELS (the latter in combination with its grounder LPARSE) must be found in the command search-path of the used system.

As depicted in Figure 5.2, spock internally transforms the parsed input program Π into a solver-compatible representation before forwarding it to the externally called answer-set solver. The resulting set of answer sets, $AS(\Pi)$, is then parsed and stored for further processing. When using flag '-o', spock outputs $AS(\Pi)$. An alternative graphical representation of the answer sets of Π, as shown in Figure 5.4, can be obtained by flag '-as'.

Command-line arguments for externally called systems can be forwarded using the flags '-dlvarg', '-lparg', and '-smarg'.

Example 5.1 Consider input file file5.1, containing program Π_{col}:

```
r1 : node(a).
r2 : node(b).
r3 : node(c).
r4 : node(d).
r5 : edge(a, b).
r6 : edge(a, c).
r7 : edge(a, d).
r8 : edge(b, c).
r9 : edge(c, d).
r10 : edge(A, B) :- edge(B, A).
r11 : colour(X, r) v colour(X, g) v colour(X, b) :- node(X).
r12 :  :- edge(A, B), colour(A, C), colour(B, C).
```

The answer sets for this program can be computed using DLV by the command:

```
java -jar spock.jar -x -o file5.1 .
```

Flag '-x' externally calls DLV on the input program and '-o' triggers the output of its answer sets.

Note that the call yields the output of the corresponding answer sets in lexicographic order. As an example, the first three answer sets are listed below, as output by spock:

```
{colour(a, b), colour(b, g), colour(c, r), colour(d, g),
edge(a, b), edge(a, c), edge(a, d), edge(b, a), edge(b, c),
edge(c, a), edge(c, b), edge(c, d), edge(d, a), edge(d, c),
node(a), node(b), node(c), node(d)}
{colour(a, b), colour(b, r), colour(c, g), colour(d, r),
edge(a, b), edge(a, c), edge(a, d), edge(b, a), edge(b, c),
edge(c, a), edge(c, b), edge(c, d), edge(d, a), edge(d, c),
node(a), node(b), node(c), node(d)}
{colour(a, g), colour(b, b), colour(c, r), colour(d, b),
edge(a, b), edge(a, c), edge(a, d), edge(b, a), edge(b, c),
edge(c, a), edge(c, b), edge(c, d), edge(d, a), edge(d, c),
node(a), node(b), node(c), node(d)}
```

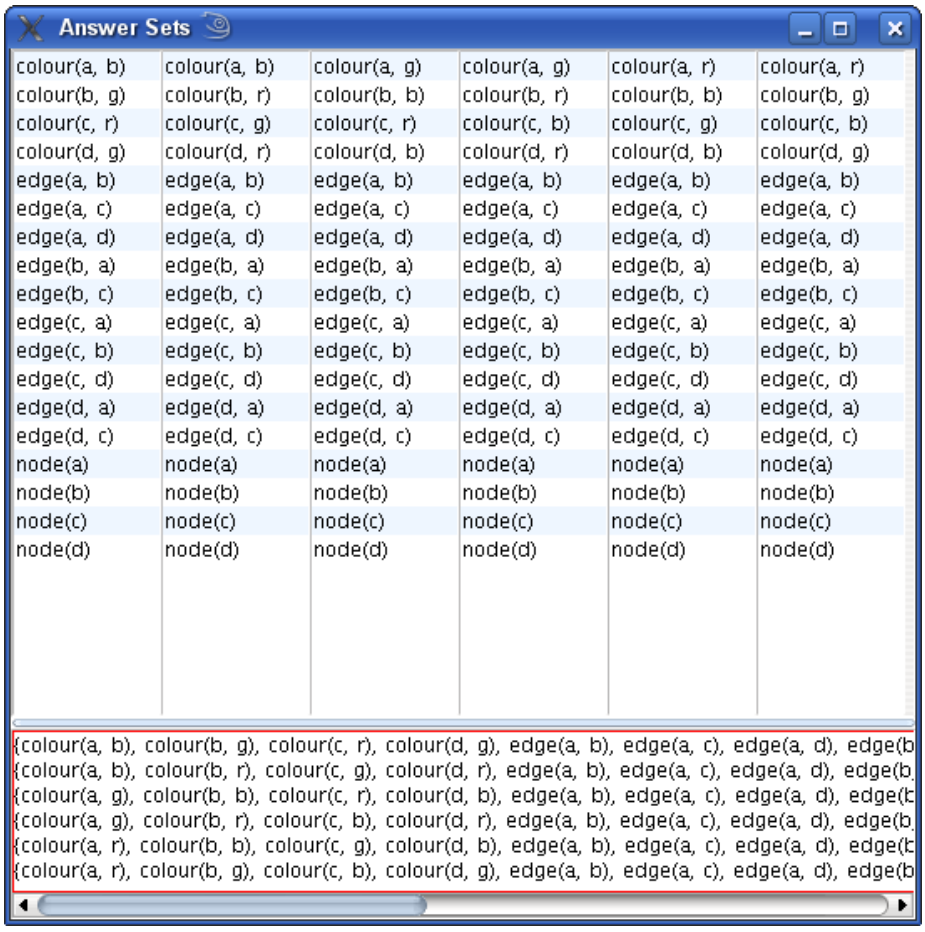

Figure 5.4: Graphical representation of $AS(\Pi_{col})$

Using the flag '`-as`', instead of '`-o`', the graphical representation of the answer sets of Π_{col}, shown in Figure 5.4, will be displayed.

Computing answer sets with `SMODELS` instead of `DLV` is usually done by a call of the form

```
java -jar spock.jar -xsm -o file5.1 ,
```

However, in this case, attempts to compute the answer sets of Π_{col} with `SMODELS` will produce an error message, as Π_{col} involves head disjunctions which are not supported by `SMODELS`. $\diamondsuit$

5.5 Meta-Program Translation

Given a propositional disjunctive logic program Π over $\mathcal{A}$, the input part, $\pi_{in(\Pi)}$, of meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ over $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$, as introduced in Chapter 3, can be obtained by the following call of `spock`:

```
java -jar spock.jar -mtr FILE ,
```

where file `FILE` contains a representation of Π.

The remaining rules of the meta-program, $\mathcal{D}_{\mathcal{M}}(\Pi) \setminus \pi_{in(\Pi)}$, can be obtained by using the flag '`-mpr`', or read from a separate input file, as they are independent of Π. A standard invocation of our debugging technique involves a second call of `spock` which reads the complete meta-program from the first system call, and computes the answer sets of $\mathcal{D}_{\mathcal{M}}(\Pi)$:

```
java -jar spock.jar -mtr -mpr FILE |
      java -jar spock.jar -x -o .
```

When reading $\mathcal{D}_{\mathcal{M}}(\Pi) \setminus \pi_{in(\Pi)}$ from a separate file `DEBUG`, which might be useful for modifying the meta-program, `DEBUG` must be read in the second call. Therefore we use the flag '`--`' to read from both, standard input and input files. The call is then as follows:

```
java -jar spock.jar -mtr FILE |
      java -jar spock.jar -x -o -- DEBUG .
```

Example 5.2 Consider program Π_{ex}, given in file `FILEex5.2`:

```
r1: a v b :- c, not d.
r2: c :- d.
r3: d v e.
r4: :- e, not a, not b, not c, not d.
```

The programmer is interested in why there is no answer set of Π_{ex} including atom e, but not a, b or c. An appropriate program over $\mathcal{A}_{\mathcal{M}}(\mathcal{A})$, stored in another file `QUERY`, can be used to express this query by means of constraints:

```
:- not int(e).
:- int(a).
:- int(b).
:- int(c).
```

For this scenario, the system call

```
java -jar spock.jar -mtr -mpr FILEex5.2 |
      java -jar spock.jar -x -o -dlvarg
      -pfilter=int,ap,bl,unsatisfied,unsupported,
      inconsistent,selfCaused QUERY
```

returns subsets of meta-answer sets, offering a projection on meta-atoms indicating the considered interpretation (`int`), rule applicability (`ap, bl`), and kind of error (`unsatisfied`, `unsupported`, `inconsistent`, `selfCaused`):

```
{ap(r2), ap(r3), bl(r1), bl(r4), int(d), int(e),
selfCaused(d), unsatisfied(r2), unsupported(d), unsupported(e)}
{ap(r2), ap(r3), bl(r1), bl(r4), int(d), int(e),
selfCaused(e), unsatisfied(r2), unsupported(d), unsupported(e)}
{ap(r2), ap(r3), bl(r1), bl(r4), int(d), int(e),
unsatisfied(r2), unsupported(d), unsupported(e)}
{ap(r3), ap(r4), bl(r1), bl(r2), int(e),
inconsistent(r4), unsatisfied(r4)}
```

♢

5.6 Tagging Approach

The kernel translation $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ of a logic program Π over $\mathcal{A}$, as presented in Chapter 4, can be obtained by the call

```
java -jar spock.jar -k FILE ,
```

where file `FILE` contains a representation of Π.

Example 5.3 For file `file5.3`, representing program Π_{K} from Example 4.3, when executing the command

```
java -jar spock.jar -k file5.3 ,
```

`spock` returns the translated program $\mathcal{T}_{\mathsf{K}}[\Pi_{\mathsf{K}}]$:

```
a :- ap(r1), not ko(r1).
ap(r1) :- ok(r1), b.
bl(r1) :- ok(r1), not b.
ok(r1) :- not -ok(r1).
a :- ap(r2), not ko(r2).
ap(r2) :- ok(r2), not c.
bl(r2) :- ok(r2), not not c.
ok(r2) :- not -ok(r2).
b :- ap(r3), not ko(r3).
ap(r3) :- ok(r3), c, not d.
bl(r3) :- ok(r3), not c.
bl(r3) :- ok(r3), not not d.
ok(r3) :- not -ok(r3).
c :- ap(r4), not ko(r4).
ap(r4) :- ok(r4), not e.
bl(r4) :- ok(r4), not not e.
ok(r4) :- not -ok(r4).
e :- ap(r5), not ko(r5).
ap(r5) :- ok(r5), a.
bl(r5) :- ok(r5), not a.
ok(r5) :- not -ok(r5).
d :- ap(r6), not ko(r6).
ap(r6) :- ok(r6), not b.
bl(r6) :- ok(r6), not not b.
ok(r6) :- not -ok(r6).
 :- falsum.
```

♢

The flags '`-expo`', '`-exco`', and '`-exlo`' activate the extrapolation translations $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, respectively. Instead of using all three flags simultaneously, setting '`-ex`' produces the union of these program translations. Since the extrapolation taggings make only sense in conjunction with the kernel tagging, we usually also use the '`-k`' flag here. In order to restrict the generation of an extrapolation tagging to a subprogram Π' of Π, the names of the considered rules must be explicitly stated in a comma-separated list following the

'`-exrules=`' flag. As programs translated via $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$ involve `SMODELS`-specific choice rules, `spock` will produce disjunctive rules, simulating the respective choice rules for use with `DLV`. However, if we want to use `SMODELS` we have to set the '`-sm`' flag to activate `SMODELS` syntax.

For computing only abnormality-minimum answer sets, as proposed in Chapter 4, `spock` makes use of `DLV`-specific weak constraints. By using the flags '`-minab`', '`-minabp`', '`-minabc`', or '`-minabl`', `spock` produces weak constraints which allow for minimising all abnormality tags, all program-oriented abnormality tags, all completion-oriented abnormality tags, or all loop-oriented abnormality tags, respectively. When using these options, we are restricted to use `DLV` as solver, as `SMODELS` cannot handle weak constraints.

Example 5.4 Consider input file `file5.4`, containing program Π_{Ex2} from Example 4.10:

```
r1: a :- not b.
r2: b :- not c.
r3: c :- not a.
```

We use the command

```
java -jar spock.jar -k -ex -exrules=r1 -sm file5.4
```

for computing the extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex2}}, \{r1\}, HB(\Pi_{\mathsf{Ex2}})]$ of Π_{Ex2} projected to $\{r1\}$ and $HB(\Pi_{\mathsf{Ex2}})$:

```
a :- ap(r1), not ko(r1).
ap(r1) :- ok(r1), not b.
bl(r1) :- ok(r1), not not b.
ok(r1) :- not -ok(r1).
b :- ap(r2), not ko(r2).
ap(r2) :- ok(r2), not c.
bl(r2) :- ok(r2), not not c.
ok(r2) :- not -ok(r2).
c :- ap(r3), not ko(r3).
ap(r3) :- ok(r3), not a.
bl(r3) :- ok(r3), not not a.
ok(r3) :- not -ok(r3).
 :- falsum.

ko(r1).
{a} :- ap(r1).
ab_p(r1) :- ap(r1), not a.
{a} :- bl(r1).
ab_c(a) :- bl(r1), a.
{b}.
ab_c(b) :- b.
{ab_l(a)} :- not ab_c(a).
a :- ab_l(a).
{ab_l(b)} :- not ab_c(b).
b :- ab_l(b).
```

For computing the abnormality-minimum answer sets of $\mathcal{T}_{\mathsf{Ex}}[\Pi_{\mathsf{Ex2}}, \{r1\}, HB(\Pi_{\mathsf{Ex2}})]$, we call `spock` as follows:

```
java -jar spock.jar -k -ex -exrules=r1 -minab file5.4 |
                java -jar spock.jar -x -o .
```

The output of this operation yields the two resulting answer sets of Example 4.10:

```
{ab_c(b), ap(r3), b, bl(r1), bl(r2), c, ko(r1), ok(r1), ok(r2), ok(r3)},
{ab_p(r1), ap(r1), ap(r3), bl(r2), c, ko(r1), ok(r1), ok(r2), ok(r3)}.
```
♢

Chapter 6

Other Approaches

In this chapter, we will describe three approaches towards debugging of answer-set programs which have recently been proposed [53, 7, 67]. All three methods deal with propositional non-disjunctive programs.

6.1 Justifications for Answer-Set Programs

Pontelli, Son, and El-Khatib [53, 26] adopt the concept of *justifications* to the context of answer-set programming. Justifications have originally been introduced as a means for debugging and understanding of PROLOG programs [58, 52, 64], by giving either a *proof description* why a certain property holds, or a *counterexample* indicating where a violation or conflict occurs in the system.

Here, the general debugging question addressed is why an atom is true or false in an answer set of a program to debug. The answers to this question are given in the form of justifications. A justification is a graph explaining the truth value of an atom with respect to a given answer set. This notion is extended by Pontelli and Son [53] to so-called *online justifications* which provide explanations also for partial and inconsistent interpretations. In both cases, the programs under consideration are propositional and normal.

Despite the fact that the computation of justifications can be elegantly done by ASP-meta-programming, the question addressed by the meta-programming approach of Chapter 3 is a different one, which makes the approaches hard to compare. In contrast to the kernel transformation of the tagging approach in Chapter 4, which also inspects answer sets of the program to debug, the information provided by a justification is not restricted to simple applicability of rules in an answer set, but delivers a full explanation for the truth of an atom a, in terms of recursive dependencies of a on other literals. The individual steps in this recursion are based on the so-called *locally consistent explanation* (LCE) which gives an immediate reason for the truth value of an atom a. The LCE is a set of sets of literals, where the truth values of the literals of such a set provide sufficient information to derive the truth value of a. If a is true with respect to interpretation I, there must be some applicable rule r such that $head(r) = \{a\}$. Then, the body of r explains the truth value of a, hence it forms an element of the LCE. If a is false with respect to I, every rule r such that $head(r) = \{a\}$ is not applicable, and there must be at least one true atom in the negative body, or at least one false atom in the positive body of r. By taking one of these atoms for every rule r such that $head(r) = \{a\}$, a member set of the LCE for the falsity of a can be built.

Definition 6.1 Let Π be a propositional normal program, $a \in HB(\Pi)$ an atom and $A \in AS(\Pi)$ an answer set of Π. Then, the *locally consistent explanation* (LCE) for a, denoted by $\sigma_\Pi(A, a)$, is a set of sets such that

- case a is true under A:

$$\sigma_\Pi(A,a) = \left\{ body(r) \;\middle|\; \begin{array}{l} r \in \Pi, \\ head(r) = \{a\}, \text{ and} \\ \text{all } l \in body(r) \text{ are true under } A \end{array} \right\},$$

- case a is false under A:

$$\sigma_\Pi(A,a) = \{\alpha_1, \ldots, \alpha_k\},$$

where each α_i is a minimal set of literals such that for all $l \in \alpha_i$, l is false under A and for all $r \in \Pi$, if $head(r) = \{a\}$ then $\alpha_i \cap body(r) \neq \emptyset$. □

Example 6.2 Consider program

$$\begin{aligned} \Pi = \{\ & a \leftarrow d, e\,, \\ & a \leftarrow c, not\ d\,, \\ & a \leftarrow not\ b\,, \\ & b \leftarrow not\ a\,, \\ & c \leftarrow not\ b\ \}. \end{aligned}$$

The program has two answer sets, $A_1 = \{a, c\}$ and $A_2 = \{b\}$. Atom a is true under A_1 and false under A_2, justified by the corresponding LCEs $\sigma_\Pi(A_1, a)$ and $\sigma_\Pi(A_2, a)$:

$$\begin{aligned} &\sigma_\Pi(A_1, a) = \{\{c, not\ d\}\}, \{not\ b\}\}; \\ &\sigma_\Pi(A_2, a) = \{\{d, c, not\ b\}, \{e, c, not\ b\}\}. \end{aligned}$$ ♢

By recursively identifying the immediate reasons of a truth value, a justification is characterised as follows:

Definition 6.3 A *justification* for an atom a with respect to a propositional normal program Π and an answer set A is a directed labelled graph $\mathcal{J}_\Pi(A, a) = \langle V, E, \{+, -\}\rangle$, where $\{a\} \subseteq V \subseteq HB(\Pi) \cup \{\top, \bot\}$ such that

1. every vertice in $V \setminus \{a\}$ is reachable from a;
2. if $b \in V$ is true under A then
   - $\emptyset \in \sigma_\Pi(A, b)$ iff $\langle b, \top, +\rangle \in E$ and $\top \in V$;
   - there is a unique $\alpha \in \sigma_\Pi(A, b)$ satisfying the conditions that for each $l \in \alpha$,
     - if l is an atom, then $l \in V$, $\langle b, l, +\rangle \in E$, $l \neq b$, and there is no path from l to b in E such that all edges in the path are labelled by $+$,
     - if l is a negated atom $not\ d$, then $d \in V$ and $\langle b, d, -\rangle \in E$;
   - there are no other outgoing edges from b;
3. if $b \in V$ is false under A then

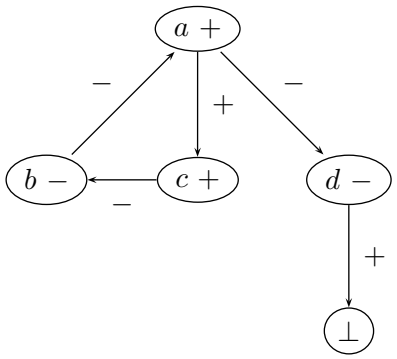

Figure 6.1: $\mathcal{J}_\Pi(A_1, a)$

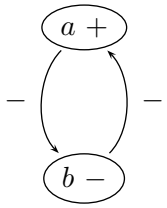

Figure 6.2: $\mathcal{J}_\Pi(A_1, a)$

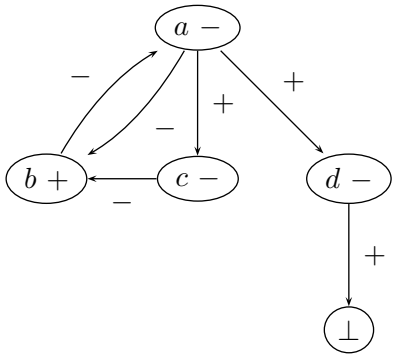

Figure 6.3: $\mathcal{J}_\Pi(A_2, a)$

answer set	atom	value	LCE
A_1	a	true	$\{\{not\ b\}, \{c, not\ d\}\}$
A_1	b	false	$\{\{not\ a\}\}$
A_1	c	true	$\{\{not\ b\}\}$
A_1	d	false	$\{\emptyset\}$
A_2	a	false	$\{\{d, c, not\ b\}, \{e, c, not\ b\}\}$
A_2	b	true	$\{\{not\ a\}\}$
A_2	c	false	$\{\{not\ b\}\}$
A_2	d	false	$\{\emptyset\}$

Table 6.1: Locally consistent explanation for atoms in A_1 and A_2

- $\emptyset \in \sigma_\Pi(A, b)$ iff $\langle b, \bot, + \rangle \in E$ and $\bot \in V$;
- there is a unique $\alpha \in \sigma_\Pi(A, b)$ such that, for each $l \in \alpha$,
  - if l is an atom, then $l \in V$ and $\langle b, l, + \rangle \in E$,
  - if l is a negated atom $not\ d$, then $d \in V$ and $\langle b, d, - \rangle \in E$.
  - there are no other outgoing edges from b; □

In other words, condition 2 states, that an atom b which is true under A has either a positive edge to $\top$, in case there is a fact $b \leftarrow$, or b has positive (respectively, negative) edges to the atoms occurring positively (respectively, negatively) in the body of a single applicable rule deriving b. There must not be any cycle of positive edges for ensuring that if b is contained in a positive loop Γ, only explanations are given that indicate external support for Γ with respect to A.

If b is false under A, condition 3 of Definition 6.3 states that there is a positive edge from b to $\bot$ if b does not occur in the head of any rule. Otherwise, for each rule r having $head(r) = \{b\}$, we have a minimum number of edges such that there is a positive (respectively, negative) edge from b to an atom (respectively, negated atom), being false under A and occurring in the body of r.

Note that Definition 6.3 is ambiguous in the sense that there may exist multiple justifications for an atom with respect to the same program and answer set.

Example 6.4 Reconsider program Π from Example 6.2. Figure 6.1 is a justification for atom a with respect to Π and A_1. As an example for the ambiguity of Definition 6.3, note that also Figure 6.2 is a justification for a with respect to Π and A_1. The graph in Figure 6.3 is a justification for a with respect to A_2.

The LCEs from which the three justifications are derived are given in Table 6.1. ♢

In the justifications of Example 6.4 we find negative cycles of explanations, for instance in Figure 6.2, a is true because b is not and b on the other hand is false because a is not false. In order to avoid this kind of situation, the authors introduced a new vertice labelled by "*assume*" [26] and used it subsequently [53] in a more refined definition for justifications. The idea behind this new vertice is to regard the truth values of atoms connected to it as given.

6.2 Query Algorithms for Debugging ASP

Brain and De Vos [7] focus on the questions why a set S of literals is satisfied by a specific answer set A and why a set S of literals is not satisfied by any answer set. The pseudocode of two procedural algorithms Λ_1 and Λ_2 is given, answering these questions for propositional normal programs. Since two problems are addressed, we will discuss them separately.

6.2.1 Why is a set of literals satisfied by a specific answer set?

Given a propositional normal program Π over $\mathcal{A}$, an answer set A of Π, and a set S of literals over $\mathcal{A}$, algorithm Λ_1 exhibits the following behaviour:

- For every atom $a \in S$, Λ_1 searches for rules r in Π such that $head(r) = \{a\}$ and r is applicable under A. For each such rule, Λ_1 returns a textual message, stating that a is in A as r is applied with respect to it.
- For every negated atom $not\ b \in S$, if there are no rules $r \in \Pi$ having $head(r) = \{b\}$, Λ_1 returns a corresponding message. Otherwise, for each (blocked) rule r such that $head(r) = \{b\}$, Λ_1 returns information about which literals in the body of r are not satisfied by A, thus stating which literals are responsible for r to be blocked.

The information gained by Λ_1 is identical to the locally consistent explanations of the elements of S, as defined in the approach by Pontelli, Son, and El-Khatib, discussed in Section 6.1. The meta-programming approach of Chapter 3 can be adapted to provide the same results by joining subprogram $\mathcal{D}_{ap}(\Pi)$ of $\mathcal{D}_{\mathcal{M}}(\Pi)$ with the rules

$$inAnyHead(At) \leftarrow head(\_, At),$$
$$lce_p(At, R) \leftarrow int(At), head(R, At), ap(R),$$
$$lce_n(At) \leftarrow atom(At), not\ inAnyHead(At),$$
$$lce_n^-(At1, R, At2) \leftarrow not\ int(At1), head(R, At1), bodyP(R, At2), not\ int(At2),$$
$$lce_n^+(At1, R, At2) \leftarrow not\ int(At1), head(R, At1), bodyN(R, At2), int(At2),$$

and with the constraints $\leftarrow not\ int(l_a)$ for all $a \in A$ and $\leftarrow int(l_b)$ for all $b \in HB(\Pi) \setminus A$, used for specifying the considered answer set. The new program has a unique meta-answer-set A_{lce} guessing A such that

- $lce_p(l_a, l_r) \in A_{lce}$ iff for rule r, $head(r) = \{a\}$ and r is applicable under A,
- $lce_n(l_a) \in A_{lce}$ iff atom $a \in HB(\Pi)$ is not in the head of any rule of Π,
- $lce_n^-(l_a, l_r, l_b) \in A_{lce}$ iff atom $a \in HB(\Pi) \setminus A$ is in the head of rule r and b is an atom in $body(r)$, unsatisfied by A, and
- $lce_n^+(l_a, l_r, l_b) \in A_{lce}$ iff atom $a \in HB(\Pi) \setminus A$ is in the head of rule r and $not\ b$ is a negated atom in $body(r)$, unsatisfied by A.

As only immediate reasons for the truth value of a literal l from S are given by algorithm Λ_1, the explanations are not very significant in the context of loops. Here, an atom a in the considered answer set is also justified by rules which do not represent the external support of a loop containing a, e.g. in program $\{r_1 = a, r_2 = a \leftarrow a\}$, the truth of a would be justified by the applicability of tautological rule r_2.

Example 6.5 Consider program

$$\begin{aligned}\Pi_{\Lambda_1} = \{\ & a \leftarrow b,\\ & c \leftarrow not\ d, a,\\ & d \leftarrow not\ c, a,\\ & b \leftarrow\ \}.\end{aligned}$$

The answer sets of Π_{Λ_1} are given by $AS(\Pi_{\Lambda_1}) = \{\{a, b, c\}, \{a, b, d\}\}$. The developer wants to know why $S = \{a, not\ \ d\}$ is satisfied by answer set $\{a, b, c\}$. Algorithm Λ_1 will return the following two answers:

> "a `is satisfied by` $\{a, b, c\}$ `because rule` $a \leftarrow b$ `is applied with respect to it.`"

> "d `is not satisfied by` $\{a, b, c\}$ `as only rule` $d \leftarrow not\ c, a$ `has head` $\{d\}$ `but` c `is in the answer set.`" ♢

6.2.2 Why is a set of literals not satisfied by any answer set?

The question, algorithm Λ_2 is dealing with, is closely related to the question why a specific interpretation is not an answer set, as addressed by our meta-programming and our extrapolation-tagging approaches. Brain and De Vos' formulation of the question suggests a more general problem, as it implies that answers do not speak about individual interpretations but subsume reasons for the specified set not being satisfied by *all* interpretations. But in general, also Λ_2 does not give a single reason for a set S of literals not to be satisfied by any answer set, but provides different explanations with respect to different classes of interpretations for the program to debug.

Given a propositional normal program Π and a set S of literals which is not satisfied by any answer set of Π, Λ_2 works in three phases:

1. The algorithm checks whether there is an atom $a \in S$ such that $\{a\}$ is not head of any rule.

2. The algorithm checks whether there is a rule $r \in \Pi$ which is applicable under the set of atoms in S such that $head(r) = \{a\}$ holds, but $not\ a$ is contained in S.

3. If no error has been found yet, Λ_2 creates multiple supersets of S in which further literals are added incrementally, until the next step would introduce an error. In the process a graph structure is built for every superset for avoiding cyclic derivations.

In phases 1 and 2, Λ_2 gives explicit answers to the question why S is not satisfied by any answer set. However, when phase 3 is reached, the programmer has to explore the created supersets, and figure out why no further literal can be added to them. In such cases Λ_2 can be seen as an algorithm, incrementally building answer sets of Π, which stops and outputs the immediate results before a partial interpretation turns out to be no answer set of Π.

6.3 Debugging Inconsistent Answer-Set Programs

Syrjänen's approach [67] addresses the issue of debugging inconsistent answer-set programs. It is adapted from the field of symbolic diagnosis [56] and designed to find reasons for the absence of answer sets. The programs under consideration are propositional, non-disjunctive, and may include choice rules. The identified reasons for inconsistency of program Π are sets of constraints in Π, and *odd loops* of Π. As shown by You and Youan [69], inconsistency is always caused by odd loops or constraints in the considered class of programs.

Debugging in Syrjänen's approach is divided into three phases. At first, odd loops are detected. If no odd loop is found, minimal sets of constraints which return consistency to the program when removed, called diagnoses, are computed. Finally, sets of constraints are searched which are called *conflict sets* and indicate constraints that cannot be satisfied at the same time.

Odd-loop detection is performed using an ASP meta-programming technique, related to the method we introduce in Chapter 3. In fact, when considering non-disjunctive programs, our meta-program $\mathcal{D}_{\mathcal{M}}(\Pi)$ can be unified with the meta-program used by Syrjänen, and thereby extend its functionality by odd-loop detection. Given inconsistent non-disjunctive programs without odd loops, our meta-programming approach can be used for computing minimal diagnosis, when minimising atoms over the *violated*/1-predicate. Syrjänen's technique for finding diagnosis bears resemblance to the tagging approach introduced in Chapter 4. For each constraint c, a new meta-atom is introduced which allows for blocking c, like it is done for all rules in tagging.

6.3.1 Odd-Loop Detection

Odd loops of a program Π are sets $\Gamma \subseteq HB(\Pi)$ of atoms where the truth value of an atom a depends recursively on itself through an odd number of negations. For example, program Π_{ex} involves an odd loop:

$$\begin{aligned}\Pi_{\text{ex}} = \{\ r_1 &= a \leftarrow \mathit{not}\ b,\\ r_2 &= b \leftarrow \mathit{not}\ c,\\ r_3 &= c \leftarrow \mathit{not}\ a\ \}.\end{aligned}$$

Assume a is true. Then, since r_3 is not applicable, we have that c is false. Thus, rule r_2 is applicable, therefore b is true, and hence a cannot be true as assumed, since r_1 is blocked. A similar contradiction arises when assuming that a is false, therefore $AS(\Pi_{\text{ex}}) = \emptyset$.

However, odd loops do not always cause inconsistency. There might be handles for an odd loop Γ, as discussed by Costantini [14], which either block rules involved in Γ or set atoms of Γ true, under some interpretation I such that $I \in AS(\Pi)$.

Therefore, odd loops might be used for pruning unwanted answer sets. Syrjänen suggests to consider all odd loops as errors, and to use a combination of choice rules and constraints for pruning.

Odd-loop detection is performed using ASP meta-programming. Given a program Π, the meta-program $\Pi_{Syr} = \mathcal{D}_{aux}(\Pi) \cup \pi_{\mathcal{M}}$ which Syrjänen uses for odd-loop detection is the union of $\mathcal{D}_{aux}(\Pi)$, as defined in Chapter 3, and program $\pi_{\mathcal{M}}$, given in Figure 6.4. The resulting answer sets include meta-atoms indicating which atoms of the inconsistent program Π are in an odd loop, and which rules $r \in \Pi$ are involved in these odd loops.

$$
\begin{aligned}
&posEdge(H, A) \leftarrow head(R, H), bodyP(R, A),\\
&negEdge(H, A) \leftarrow head(R, H), bodyN(R, A),\\
&even(X, Y) \leftarrow posEdge(X, Y),\\
&odd(X, Y) \leftarrow negEdge(X, Y),\\
&even(X, Z) \leftarrow posEdge(X, Y), even(Y, Z), atom(Z),\\
&odd(X, Z) \leftarrow posEdge(X, Y), odd(Y, Z), atom(Z),\\
&odd(X, Z) \leftarrow negEdge(X, Y), even(Y, Z), atom(Z),\\
&even(X, Z) \leftarrow negEdge(X, Y), odd(Y, Z), atom(Z),\\
&oddLoop(X) \leftarrow odd(X, X),\\
&inOddLoop(X, Y) \leftarrow odd(X, Y), even(Y, X),\\
&firstInLoop(A) \leftarrow oddLoop(A), not\ hasPredecessor(A),\\
&hasPredecessor(A) \leftarrow inOddLoop(B, A), B < A,\\
&ruleInLoop(R, Z) \leftarrow inOddLoop(X, Y), inOddLoop(X, Z),\\
&\qquad\qquad firstInLoop(Z), head(R, X), bodyP(R, Y),\\
&ruleInLoop(R, Z) \leftarrow inOddLoop(X, Y), inOddLoop(X, Z),\\
&\qquad\qquad firstInLoop(Z), head(R, X), bodyN(R, Y).
\end{aligned}
$$

Figure 6.4: Subprogram $\pi_{\mathcal{M}}$ of Π_{Syr}

6.3.2 Finding Inconsistency-Causing Constraints

Considering only programs without odd loops, inconsistency is related to applicable integrity constraints [69]. The main goal is to find sets of constraints in program Π, called *diagnoses*, whose removal from Π returns consistency to Π.

Definition 6.6 Let Π be a propositional non-disjunctive program without odd loops. Then, a *diagnosis of* Π is a set $D \subseteq \mathfrak{C}(\Pi)$ of constraints such that $\Pi \setminus D$ is consistent. □

Consider example program

$$
\begin{aligned}
\Pi_{\text{ex}} = \{\ r_1 &= a \leftarrow not\ e,\\
r_2 &= e \leftarrow not\ a,\\
c_1 &= \leftarrow a,\\
c_2 &= \leftarrow not\ a,\\
c_3 &= \leftarrow not\ b\ \}.
\end{aligned}
$$

This program has three diagnoses: $D_1 = \{c_1, c_3\}, D_2 = \{c_2, c_3\}$, and $D_3 = \{c_1, c_2, c_3\}$. In order to reduce the amount of debugging information, Syrjänen considers only cardinality-minimal diagnoses. In case of program Π_{ex}, D_1 and D_2 are minimal diagnoses, where $|D_1| = |D_2| = 2$.

Diagnoses are detected using optimisation techniques of the answer-set solver. Each constraint $c = \leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, a_n$ in the original program Π is replaced by two new rules:

$$
\begin{aligned}
&\leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, a_n, not\ remove(c), \quad \text{and}\\
&constraint(c) \leftarrow .
\end{aligned}
$$

The atom *remove*(c) offers a handle to block constraint c. Using SMODELS' choice rules, cardinality constraints and conditional literals, a fixed number of *remove*-atoms can be guessed. Starting with a single one, by successively increasing the number of removed constraints, the solving-procedure is repeated until answer sets exist. The *remove*-atoms in these answer sets then indicate the minimal diagnoses of Π.

The same effect could be achieved in DLV, by using a disjunctive guess

$$\mathit{remove}(c) \vee \overline{\mathit{remove}}(c) \leftarrow$$

for each constraint c, and a weak-constraint for minimising atoms over the *remove*/1-predicate.

In order to identify interrelations between the constraints of a program, Syrjänen introduces the notion of *conflict sets*. A set of constraints is a conflict set if every minimal diagnosis of the program contains exactly one member from the set.

Definition 6.7 Let Π be a propositional non-disjunctive program without odd loops and $\mathfrak{D}(\Pi)$ the set of its minimal diagnoses. Then, a *conflict set* is a set $C \subseteq \mathfrak{C}(\Pi)$ of constraints such that

1. for all diagnoses $D \in \mathfrak{D}(\Pi)$ it holds that $|D \cap C| = 1$; and
2. for all constraints $c \in \mathfrak{C}(\Pi)$ there exists a diagnosis $D \in \mathfrak{D}(\Pi)$ such that $c \in D$. □

For program Π_{ex} from above, there are the two conflict sets, $\{c_1, c_2\}$ and $\{c_3\}$. Intuitively, each conflict set represents constraints that cannot be fullfilled at the same time. Note that conflict sets do not necessarily exist.

Like diagnoses, conflict sets are computed using SMODELS-specific constructs and an iterated invocation of the solving procedure. During this iteration the number of estimated conflict sets is increased until conflict sets are found, or we know that none exist. The maximal number of iteration steps is limited by the number of constraints.

Chapter 7

Conclusion

In this work we have dealt with the issue of debugging propositional answer-set programs. Debugging in ASP has not been studied thoroughly yet, but is important for practical application of logic programs under the answer-set semantics.

We have introduced two new techniques for debugging propositional answer-set programs. One of these approaches is based on meta-programming, lifting a given propositional disjunctive program to the language of a non-ground meta-program. Our technique tackles the question why specific interpretations are not answer sets of a program to debug. The classification of reasons why an interpretation is no answer set is based on an alternative characterisation of the answer-set semantics. As meta-programming is very powerful and flexible, our method can be adapted to address many debugging relevant questions.

The other method introduced is a tagging technique which augments a non-disjunctive program to debug with dedicated atoms for controlling the applicability of rules and analysing the program. There are different versions of the method, addressing different debugging questions. In the basic variant, the debugging system gives statements about the applicability of rules in context of answer sets of the program to debug. An extended version of the technique which is related to the meta-programming method, searches for abnormalities causing an interpretation not to be an answer set.

Both approaches are declarative and therefore independent of the algorithm for computing the answer sets of a program. Moreover, in both methods, the results of the debugging process can be read off the answer sets of a debugging program. In order to reduce the amount of debugging information to relevant parts, we make use of standard optimisation techniques of ASP.

Besides our techniques, we have discussed and compared three existing approaches towards debugging of non-disjunctive propositional answer-set programs. Furthermore, the translations, needed for the introduced techniques have been implemented in the prototype debugging-tool `spock`.

Very important for future research into debugging of answer-set program is to handle non-ground programs, as programs in real-world application are typically non-ground. Clearly, debugging strategies for propositional programs can be applied to the grounding of a non-ground program. In general, however, groundings are of huge size. Thus, the problem of relating non-ground rules to their ground instances is a major challenge here. Moreover, another difficulty caused by the enormous size of groundings is keeping the debugging process efficient. Another open task is investigating the specifics of restricting debugging to certain

program modules. Furthermore, future implementations for debugging should provide easy-to-use interfaces, and be embedded in integrated development environments.

Appendix

Selected Argument Options of `spock`

`--`	If a filename is given, `spock` does not read from standard input, unless this flag is set.
`-p`	Outputs the given program with rule labels.
`-c`	Outputs the given program without rule labels.
`-x`	Runs `DLV` on the given program.
`-xsm`	Runs `Smodels` on the given program.
`-n=NR`	Computes maximally *NR* many answer sets.
`-sm`	Formats various output in `Smodels` syntax, otherwise `DLV` syntax is used.
`-o`	Outputs all computed or read answer sets.
`-as`	Displays all computed or read answer sets in a GUI frame.
`-mtr`	Outputs meta-translation $\pi_{in(\Pi)}$.
`-mpr`	Outputs residual meta-program $\mathcal{D}_{\mathcal{M}}(\Pi) \setminus \pi_{in(\Pi)}$.
`-k`	Outputs the kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi]$ of a given program Π.
`-ex`	Outputs the extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi, HB(\Pi)]$ of a given program Π (like `-expo -exco -exlo`; see next).
`-expo`	Outputs the program-oriented extrapolation tagging $\mathcal{T}_{\mathsf{P}}[\Pi]$ of a given program Π.
`-exco`	Outputs the completion-oriented extrapolation tagging $\mathcal{T}_{\mathsf{C}}[\Pi, HB(\Pi)]$ of a given program Π.
`-exlo`	Outputs the loop-oriented extrapolation tagging $\mathcal{T}_{\mathsf{L}}[HB(\Pi)]$ of a given program Π.
`-exrules=r,s,...`	Restricts extrapolation tagging generation to rules labelled `r`, `s`, ...
`-minab`	Outputs weak constraints to minimise abnormality tags (like the ones described next).
`-minabp`	Outputs weak constraints to minimise program-oriented abnormality tags.

`-minabc`	Outputs weak constraints to minimise completion-oriented abnormality tags.
`-minabl`	Outputs weak constraints to minimise loop-oriented abnormality tags.
`-koall`	Outputs atom `ko(`n_r`)` for every rule r in the given program.
`-nas`	Outputs the number of computed or read answer sets.
`-dlvarg ARG`	Adds an argument for external calls of `DLV`.
`-lparg ARG`	Adds an argument for external calls of `LPARSE`.
`-smarg ARG`	Adds an argument for external calls of `SMODELS`.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[2] ANSI/IEEE. *Standard Glossary of Software Engineering Terminology.* IEEE, New York, NY, USA, 1983.

[3] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, USA, 1988.

[4] C. Baral and M. Gelfond. Reasoning Agents in Dynamic Domains. In J. Minker, editor, *Proceedings of the Workshop on Logic-Based Artificial Intelligence, (LBAI'99), Washington, DC, USA*. Computer Science Department, University of Maryland, 1999.

[5] M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. Debugging ASP Programs by Means of ASP. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), Tempe, AZ, USA*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 31–43. Springer, 2007.

[6] M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. "That is Illogical Captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description. In M. De Vos and T. Schaub, editors, *Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming (SEA'07), Tempe, AZ, USA*, pages 71–85, 2007.

[7] M. Brain and M. D. Vos. Debugging Logic Programs under the Answer-Set Semantics. In M. D. Vos and A. Provetti, editors, *Answer-Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd International Answer-Set Programming Workshop, (ASP'05), Bath, England, UK*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[8] F. Buccafurri and G. Caminiti. A Social Semantics for Multi-agent Systems. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05), Diamante, Italy*, volume 3662 of *Lecture Notes in Computer Science*, pages 317–329. Springer, 2005.

[9] F. Buccafurri and G. Gottlob. Multiagent Compromises, Joint Fixpoints, and Stable Models. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming*

and Beyond, Essays in Honour of Robert A. Kowalski, Part I, volume 2407 of *Lecture Notes in Computer Science*, pages 561–585. Springer, 2002.

[10] "bug." Britannica Concise Encyclopedia. Encyclopædia Britannica Online, 18 2007. http://www.britannica.com/ebc/article-9358233.

[11] P. Burek and R. Grabos. Dually Structured Concepts in the Semantic Web: Answer Set Programming Approach. In A. Gómez-Pérez and J. Euzenat, editors, *The Semantic Web: Research and Applications, Proceedings of the 2nd European Semantic Web Conference, (ESWC'05), Heraklion, Crete, Greece*, volume 3532 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2005.

[12] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–324. Plenum Press Publishing, New York, NY, USA, 1978.

[13] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, New York, NY, USA, 1981.

[14] S. Costantini. Comparing Different Graph Representations of Logic Programs under the Answer-Set Semantics. In A. Provetti and T. C. Son, editors, *Answer-Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st International Answer-Set Programming Workshop, (ASP'01), Stanford, CA, USA*, 2001.

[15] J. Delgrande, T. Schaub, and H. Tompits. A Framework for Compiling Preferences in Logic Programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.

[16] J. Dix, U. Kuter, and D. Nau. Planning in Answer-Set Programming using Ordered Task Decomposition. In B. N. A. Günther, R. Kruse, editor, *Proceedings of the 27th German Annual Conference on Artificial Intelligence, (KI'03), Hamburg, Germany*, pages 490–504. Springer, 2003.

[17] U. Egly, T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Computing Stable Models with Quantified Boolean Formulas: Some Experimental Results. In A. Provetti and T. C. Son, editors, *Answer-Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st International Answer-Set Programming Workshop, (ASP'01), Stanford, CA, USA*, 2001.

[18] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[19] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV system. In J. Minker, editor, *Logic-based Artificial Intelligence*, pages 79–103. Kluwer Academic Press, Norwell, MA, USA, 2000.

[20] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer-Set Programming. *Theory and Practice of Logic Programming*, 3(4):463–498, 2003.

[21] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. The DLV$^{\mathrm{K}}$ Planning System: Progress Report. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of*

the 8th European Conference on Logics in Artificial Intelligence (JELIA'02), Cosenza, Italy, volume 2424 of *Lecture Notes in Computer Science*, pages 541–544, 2002.

[22] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[23] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons and Benchmarks. In G. Cohn, L. Schubert, and S. Shapiro, editors, *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning, (KR'98), Trento, Italy*, pages 406–417. Morgan Kaufmann, 1998.

[24] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer-set Programming with Description Logics for the Semantic Web. In D. Dubois, C. A. Welty, and M.-A. Williams, editors, *Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning, (KR'04), Whistler, Canada*, pages 141–151. AAAI Press, 2004.

[25] T. Eiter and A. Polleres. Towards Automated Integration of Guess and Check Programs in Answer-Set Programming: a Meta-Interpreter and Applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006.

[26] O. El-Khatib, E. Pontelli, and T. C. Son. Justification and Debugging of Answer-Set Programs in ASP. In C. Jeffery, J.-D. Choi, and R. Lencevicius, editors, *Proceedings of the 6th International Workshop on Automated Debugging, (AADEBUG'05), Monterey, CA, USA*, pages 49–58. ACM, 2005.

[27] Free Software Foundation Inc. GNU General Public License - Version 2, June 1991, 1991. http://www.gnu.org/copyleft/gpl.html.

[28] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized Algorithmic Debugging and Testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming-Language Design and Implementation, (PLDI'91), Toronto, Canada*, pages 317–326. ACM Press, 1991.

[29] M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics. In D. Seipel, M. Hanus, A. Wolf, and J. Baumeister, editors, *Proceedings of the 21st Workshop on (Constraint) Logic Programming, (WLP'07), Würzburg, Germany*, pages 258–261. Technical Report 434, Bayerische Julius-Maximilians-Universität Würzburg, Institut für Informatik, 2007.

[30] A. V. Gelder, K. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[31] M. Gelfond and J. Galloway. Diagnosing Dynamic Systems in A Prolog. In A. Provetti and T. C. Son, editors, *Answer-Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st International Answer-Set Programming Workshop, (ASP'01), Stanford, CA, USA*, 2001.

[32] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming, (ICLP'88), Seattle, WA, USA*, pages 1070–1080. The MIT Press, 1988.

[33] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[34] A. M. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, England, UK, 1985.

[35] K. Heljanko and I. Niemelä. Answer-Set Programming and Bounded-Model Checking. In A. Provetti and T. C. Son, editors, *Proceedings of the AAAI Spring 2001 Symposium on Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, Stanford, USA*, pages 90–96. AAAI Press, Technical Report SS-01-01, 2001.

[36] K. Inoue and C. Sakama. Negation as Failure in the Head. *Journal of Logic Programming*, 35(1):39–78, 1998.

[37] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[38] R. A. Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979.

[39] J. Lee. A Model-Theoretic Counterpart of Loop Formulas. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence, (IJCAI'05), Edinburgh, Scotland, UK*, pages 503–508. Professional Book Center, 2005.

[40] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[41] Y. Lierler. CMODELS - SAT-Based Disjunctive Answer-Set Solver. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05), Diamante, Italy*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2005.

[42] V. Lifschitz. Action Languages, Answer Sets and Planning. In K. R. Apt, D. S. Warren, and M. Truszczynski, editors, *The Logic Programming Paradigm: A 25-YEAR Perspective*, pages 257–373. Springer, Secaucus, NJ, USA, 1999.

[43] V. Lifschitz and H. Turner. Splitting a Logic Program. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming, (ICLP'94), Santa Margherita Ligure, Italy*, pages 23–38. MIT-Press, 1994.

[44] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[45] V. W. Marek and J. B. Remmel. On the Expressibility of Stable Logic Programming. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR'01), Vienna, Austria*, volume 2173 of *Lecture Notes in Computer Science*, pages 107–120. Springer, 2001.

[46] V. W. Marek and M. Truszczynski. Stable Models and an Alternative Logic Programming Paradigm. In K. R. Apt, D. S. Warren, and M. Truszczynski, editors, *The Logic Programming Paradigm: A 25-YEAR Perspective*, pages 375–398. Springer, Secaucus, NJ, USA, 1999.

[47] L. Naish. Declarative Debugging of Lazy Functional Programs. In *Proceedings of the 4th Workshop on Logic Programming Environments, (WLPE'92), Washington, DC, USA*, pages 29–34, 1992.

[48] I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In I. Niemelä and T. Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning, (CNMR'98), Trento, Italy*, pages 72–79, 1998.

[49] D. V. Nieuwenborgh, M. D. Vos, S. Heymans, and D. Vermeir. Hierarchical Decision Making in Multi-agent Systems Using Answer-Set Programming. In K. Inoue, K. Satoh, and F. Toni, editors, *Proceedings of the 7th International Workshop on Computational Logic in Multi-Agent Systems, (CLIMA VII), Hakodate, Japan*, volume 4371 of *Lecture Notes in Computer Science*, pages 20–40. Springer, 2006.

[50] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A Prolog decision support system for the Space Shuttle. In A. Provetti and T. C. Son, editors, *Answer-Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st International Answer-Set Programming Workshop, (ASP'01), Stanford, CA, USA*, 2001.

[51] M. Osorio, J. A. Navarro, and J. Arrazola. Debugging in A-Prolog: A Logical Approach. In *Proceedings of the 18th International Conference on Logic Programming, (ICLP'02), London, England, UK*, pages 482–483. Springer-Verlag, 2002.

[52] G. Pemmasani, H.-F. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online Justification for Tabled Logic Programs. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, Proceedings of the 7th International Symposium on Functional and Logic Programming, (FLOPS'04), Nara, Japan*, volume 2998 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2004.

[53] E. Pontelli and T. C. Son. Justifications for Logic Programs Under Answer Set Semantics. In S. Etalle and M. Truszczynski, editors, *Proceedings of the 22nd International Conference on Logic Programming, (ICLP'06),Seattle, WA, USA*, volume 4079 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2006.

[54] R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press Publishing, New York, NY, USA, 1978.

[55] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[56] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[57] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[58] A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying Proofs Using Memo Tables. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, (PPDP'00), Montreal, Canada*, pages 178–189, 2000.

[59] B. Ruzicka. Entwurf und Implementierung eines Debugger in einer Expertensystemumgebung. Diplomarbeit, Vienna University of Technology, Vienna, Austria, September 1990.

[60] R. Schindlauer. *Answer-Set Programming for the Semantic Web*. Dissertation, Vienna University of Technology, Vienna, Austria, December 2006.

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

[62] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1):181–234, 2002.

[63] T. Soininen and I. Niemelä. Developing a Declarative Rule Language for Applications in Product Configuration. In G. Gupta, editor, *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages, (PADL'99), San Antonio, TX, USA*, pages 305–319. Springer, 1999.

[64] G. Specht. Generating Explanation Trees even for Negations in Deductive Database Systems. In *Proceedings of the 5th Workshop on Logic Programming Environments, (WLPE'93) , Vancouver, Canada*, pages 8–13, Vancouver, Canada, 1993.

[65] T. Syrjänen. *Lparse 1.0 User's Manual*. Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland. http://www.tcs.hut.fi/Software/smodels.

[66] T. Syrjänen. Implementation of Local Grounding for Logic Programs With Stable Model Semantics. Technical Report B18, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, October 1998.

[67] T. Syrjänen. Debugging Inconsistent Answer-Set Programs. In J. Dix and A. Hunter, editors, *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning, (NMR'06), Lake District, England, UK*, pages 77–83. University of Clausthal, Department of Informatics, Technical Report, IfI-06-04, 2006.

[68] H. Wertz. Stereotyped Program Debugging : an Aid for Novice Programmers. *International Journal of Man-Machine Studies*, 16(4):379–392, 1982.

[69] J.-H. You and L. Y. Yuan. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences*, 49(2):334–361, 1994.

Index