

INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

AN INTRODUCTION TO ANSWER SET PROGRAMMING AND ITS APPLICATIONS

Jakob Pippig

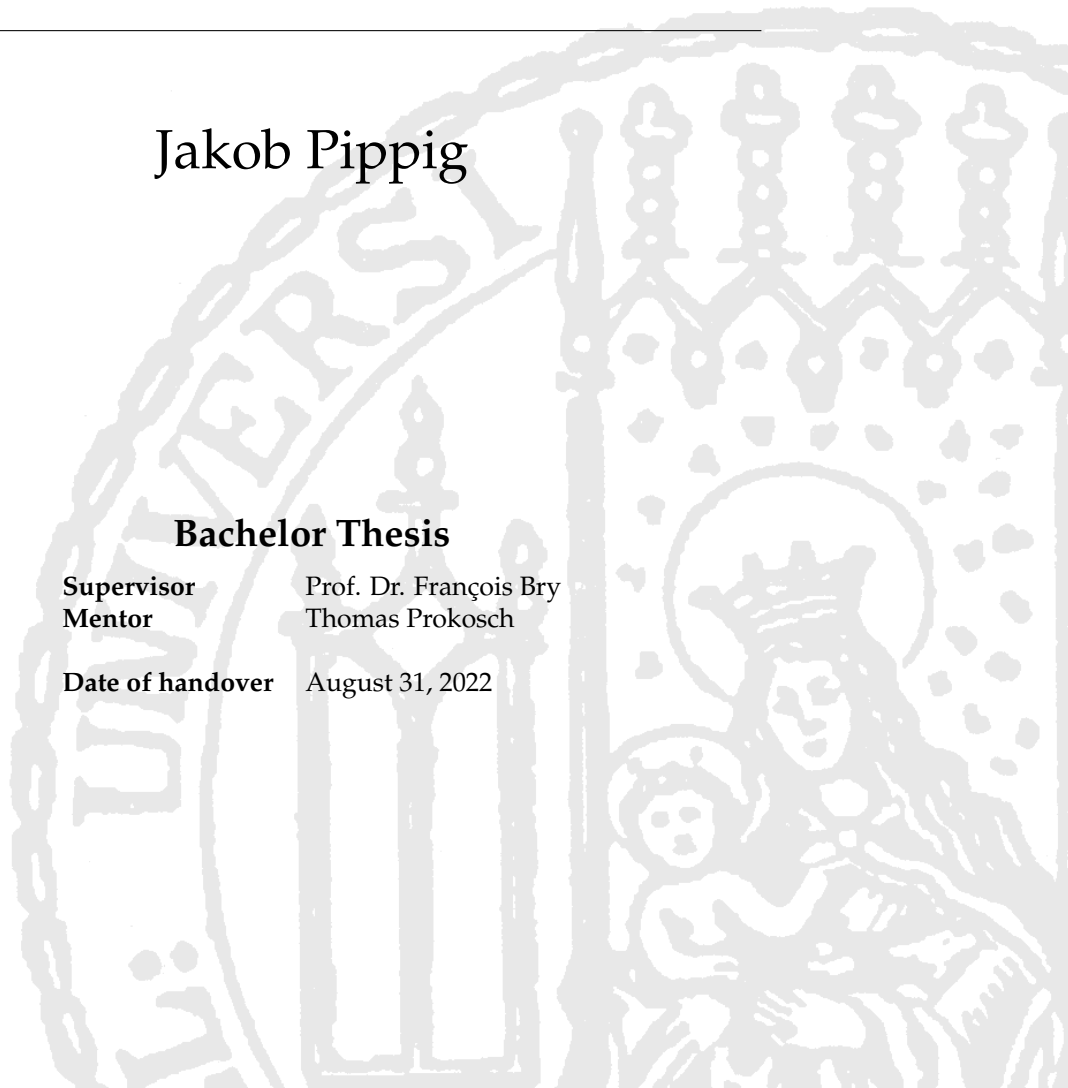
Bachelor Thesis

Supervisor
Mentor

Prof. Dr. François Bry
Thomas Prokosch

Date of handover

August 31, 2022



Declaration

I herewith declare that I wrote this thesis on my own and did not use any other sources or any other help than those mentioned.

A handwritten signature in black ink, reading "J. Pippig". The signature is written in a cursive style with a large, stylized 'J' and a long, sweeping underline.

München, August 31, 2022

Jakob Pippig

Abstract

Answer set programming is a branch of logic programming utilizing the stable model semantics to compute solutions for modelled problems. This thesis offers an overview which highlights important scientific works that inspired, created, and advanced answer set programming. The foundations of first-order logic used by answer set programming as well as the stable model semantics will be briefly described. The programming language AnsProlog will be explained and demonstrated with the help of multiple examples. The systems used to implement answer set programming will be described, showing the evolution of older approaches into the current standards. An exemplary use of answer set programming will be outlined.

Zusammenfassung

Answer Set Programming ist ein Zweig der Logikprogrammierung, der die stabile Modellsemantik zur Berechnung von Lösungen für Probleme nutzt. Diese Arbeit bietet einen Überblick über die wichtigsten wissenschaftlichen Arbeiten, welche Answer Set Programming inspiriert, geschaffen und weiterentwickelt haben. Die Grundlagen der Logik erster Ordnung, die von Answer Set Programming verwendet wird, sowie die stabile Modellsemantik werden umrissen. Die Funktionsweise des Answer Set Programming und ihre Implementierung in der Programmiersprache AnsProlog wird erläutert und anhand mehrerer Beispiele demonstriert. Die Systeme, die zur Implementierung von Answer Set Programming verwendet werden, werden beschrieben, wobei die Entwicklung von älteren Ansätzen zu den heutigen Standards aufgezeigt wird. Eine beispielhafte Anwendung von Answer Set Programming wird umrissen.

Acknowledgments

I would like to express my gratitude to professor François Bry. I would also like to thank my mentor, Thomas Prokosch, for helping me and guiding me through this project.

Contents

1	Introduction	1
1.1	Aim of this thesis	2
1.2	Related work: A short history of answer set programming	3
2	Concepts of answer set programming	5
2.1	Logic programming: Concepts and notations	6
2.2	Stable model semantics in answer set programming	9
2.3	The paradigm of answer set programming	12
3	Answer set programming with AnsProlog	16
3.1	Syntax and semantics	17
3.2	Grounding	22
4	Practical Aspects of Answer Set Programming	26
4.1	Smodels and lparse	27
4.2	Clasp and GrinGo	30
4.3	A decision support system for a Space Shuttle - An example	34
5	Conclusion	36
5.1	Summary	37

Code Examples

3.1	Paths in a directed graph	17
3.2	Coloured graph nodes	18
3.3	Negations	18
3.4	Choice rule	19
3.5	Added choice rule	19
3.6	Timed aggregate	19
3.7	Abbreviations	20
3.8	Cardinality constraint	20
3.9	Weight rule	20
3.10	Grounding size	22
3.11	Instantiating	22
3.12	Instantiating of a rule	23
3.13	Instantiating of a program	23
3.14	Recursive instantiation	24
3.15	Grounded recursive rule	25
4.1	Domain-restricted program	27
4.2	<i>Smodels</i> algorithm	28
4.3	Relevant and irrelevant variables	31
4.4	Nogoods	32
4.5	Pressurized node	35

CHAPTER 1

Introduction

Declarative programming is an alternative approach to imperative programming where instead of computing how the problem should be solved a model of the logic behind the problem is constructed and solved. A well known representative of the declarative approach to programming is logic programming. The idea of logic programming is to use mathematical logic to represent and execute computer programs.

One particular variety of logic programming is answer set programming. Answer set programming is rooted in relational databases, non-monotonic reasoning, knowledge representation, and model theory. It uses and combines the strengths of these subject areas to easily model and solve combinatorial or optimization problems.

Over the years many ideas have been developed to both make answer set programming more accessible as well as to increase the scope and types of problems it can solve. Examples for this include aggregate and recursive functions, double negation, as well as partial functions. Programming languages like AnsProlog were created to yield a programming framework which utilizes answer set semantics.

Answer set programming is a powerful tool that can be used in a multitude of ways reaching from simple planning and optimization tasks up to solving highly complex DNA analyses.

1.1 Aim of this thesis

Answer set programming is seldomly used outside of academical endeavours even though it offers wide applicability. The goal of this thesis is not only to show possible and actual uses of answer set programming but also to convey the ideas and principles behind it.

- This thesis explains and defines the necessary concepts and terminologies required to construct answer set programs. This includes the basics of logic programming, in particular first-order logics as well as the stable model semantics. Examples will illustrate these concepts.
- The programming language AnsProlog will be explained. It will be shown how the components of answer set programming are implemented and used in AnsProlog as well as the steps AnsProlog has to take to derive an answer for an answer set program. With AnsProlog being the de-facto standard for answer set programming, this thesis also highlights how AnsProlog came to be the way it is.
- This is being followed by concrete implementations of AnsProlog, outlining the practical aspects of deriving solutions for problems. The differences of some individual AnsProlog implementations are briefly discussed as well.
- Finally, an actual use case of answer set programming is showcased and it is explained why answer set programming is attractive for the problem in the first place.

In the following sections the history of answer set programming will be outlined to provide context for some of the evolutionary steps of answer set programming. Additionally, the most relevant works will be highlighted.

1.2 Related work: A short history of answer set programming

Logic programming is a programming paradigm based on a use of mathematical logic to execute and depict computer programs. Logic programming is counted towards the declarative programming languages. Declarative programming means to solve a problem by simply stating the problem. An early declarative programming language is Prolog which was developed by Alain Colmerauer in 1972. Prolog enables programmers to use first-order logics to execute computer programs [40]. It forms the basis for many modern and still widely used logic programming languages including XSB, Visual Prolog, and A-Prolog. In 1976, Robert Kowalski standardized the semantics of predicate logic for the use in programming languages with his colleague Van Emden [64].

Some concepts of answer set programming have been introduced and formalized in the following years. In 1977, Raymond Reiter published a paper explaining the idea of closed-world assumption in databases, one of the concepts which is utilized by answer set programming [54]. One year later, Keith Clark wrote about negation as failure, which plays a vital part in how answer set programming views information [19].

Raymond Reiter continued to advance logic programming and introduced the concept of non-monotonic reasoning ten years later in 1987 [53].

The stable model semantics has been investigated by Michael Gelfond and Vladimir Lifschitz in 1988 and is one of the core ideas behind answer set programming [34].

The stable model semantics can be used to effectively solve combinatorial problems which is why it seemed appealing to utilize it to deal with problems such as plan generation and knowledge representation. Ilkka Niemelä and Victor Marek also contributed to the evolution of the stable model.

Niemelä focused on knowledge configuration, an activity which is part of product configuration, where a product is customized to fit the need of a particular customer [61]. Marek suggested to use stable model semantics for planning problems [63]. Both authors suggest the use of answer set solvers to determine the stable model as a solution for their respective problems.

Because of the many conceptual additions to logic programming, in particular by making use of the stable model semantics, it was suggested by Lifschitz in his 1999 book "Action Languages, Answer Sets, and Planning", that the name for this new problem solving paradigm should be "answer set programming" [43]. This suggestion was promptly adopted and as such answer set programming became a new branch of logic programming. Various tools and programs to apply answer set programming to a problem have been developed. The most notable early example of such a tool is *Smodels*, a program which takes a set of grounded rules as input and computes the possible stable models for these. An improved version, published in 1999, offers features such as cardinality and weight constraints which further increases expressibility [58].

The grounder *lpars* was developed together with *Smodels*, its sole purpose is to replace variables in sets of rules given as input such that *Smodels* could compute their answer. These two tools form the basis for many other implementations such as *ASSAT*, *Smodels-cc*, and *GnT*. Two implementations of grounding and answer set solving that follow a different approach are *Cmodels* and *Clasp*.

Loosely based on *Smodels* the program *Cmodels* was presented by Lierler in 2005 [42]. Its approach uses incremental answer set solving which allows for partial evaluation of a program. The program *Clasp* has been developed in 2007 by Gebser, Neumann, Kaufmann, and Schaub [28]. It offers a conflict-driven approach to answer set programming, introducing *nogoods*, a type of constraint applied to an answer set program which is used to determine non-relevant parts of a rule.

The number of applications of answer set programming outside of academia is low. One exception to this is an application prototype that has been developed by the National Aeronautics and Space Administration which is a decision support system for a space shuttle [51].

Answer set programming is still evolving. Some of the advancements that were made include reactive answer set programming [24], optimization constructs [60], and quantified answer set programming [4], to only name a few.

An event contributing to the evolution of answer set programming is the open answer set programming competition, a biennially benchmark where the different answer set solvers are compared to each other [12]. These competitions not only set a standard for how answer set solvers work but also present possible different approaches to the solving process [30].

CHAPTER 2

Concepts of answer set programming

Answer set programming is a declarative programming paradigm to model and solve a variety of problems. Declarative programming states the problem and then proceeds to find an answer. Logic programming is the basis for answer set programming.

To explain answer set programming and its foundation, the stable model semantics, it is important to explain first-order logic and model theory, which is done in the following chapters.

2.1 Logic programming: Concepts and notations

Definition 1 (Logic symbols): The elements a first-order logic are the logic connectives \wedge (conjunction), \vee (disjunction), \rightarrow (implication), \neg (negation), the quantifiers \forall (universal quantifier) and \exists (existential quantifier), as well as an infinite amount of variables x, y, z, x_1, y_1, \dots , the closing and opening parentheses $)$, $($, and the truth values \perp (false) and \top (true) [8].

Definition 2 (Signature): The signature L of a first-order logic can be described as a pair of sets $L = (\{Funk_L^n\}_{n \in \mathbb{N}}, \{Rela_L^n\}_{n \in \mathbb{N}})$ where $Funk_L^n$ and $Rela_L^n$ are two families of computably enumerable symbol sets, the n -ary functions and the n -ary relations of L . The 0-ary functions are called constants and the 0-ary relations are called propositional relation symbols.

Definition 3 (Term): Let L be a signature. An L -term is inductively defined as follows:

1. Each variable is an L -term.
2. Each constant in L is an L -term.
3. Let $n \geq 1$, f be an n -ary function symbol, and t_1, \dots, t_n be L -terms. Then $f(t_1, \dots, t_n)$ is an L -term.

Definition 4 (Atoms): Let L be a signature, $n \in \mathbb{N}$, p be an n -ary relation of L and t_1, \dots, t_n be L -terms. Then $p(t_1, \dots, t_n)$ is an L -atom. If $n = 0$ then the atom $p()$ is called a propositional L -atom and is written p [15].

Definition 5 (Literal and negation): Let A be an atom. Then A and $\neg A$ are literals [5].

Definition 6 (Formula): Let L be a signature. An L -formula is inductively defined as follows:

1. Every L -atom is an L -formula.
2. \top and \perp are L -formulas.
3. If φ is an L -formula then $\neg\varphi$ is also an L -formula.
4. If φ and ϕ are L -formulas then $(\varphi \wedge \phi)$, $(\varphi \vee \phi)$, and $(\varphi \rightarrow \phi)$ are also L -formulas.
5. If φ is an L -formula and x is a variable then $\forall x\varphi$ and $\exists x\varphi$ are also L -formulas.

Definition 7 (Subformula): Let φ and ϕ be formulas. An immediate subformula is inductively defined as follows:

Atomic formulas and the truth values \top and \perp have no immediate subformulas.

The formulas $(\varphi \wedge \phi)$, $(\varphi \vee \phi)$, and $(\varphi \rightarrow \phi)$ each have the two immediate subformulas: φ and ϕ .

The formulas $\exists x\varphi$, $\forall x\varphi$ and $\neg\varphi$ each have the immediate subformula: φ .

The subformulas of φ are φ itself as well as all immediate subformulas of φ and the subformulas of the immediate subformulas of φ .

Example 1: Let $\varphi = (\neg(\top \wedge \perp) \rightarrow \top)$ be a formula. Its immediate subformulas are \top and $\neg(\top \wedge \perp)$. $\neg(\top \wedge \perp)$ has the immediate subformula $(\top \wedge \perp)$. The formula $(\top \wedge \perp)$ has two

immediate subformulas \top and \perp .

Definition 8 (Clause): A clause is a disjunction of a finite set of literals. Let A_1, \dots, A_n be a set of atoms, n and m be arbitrary numbers with $n \geq 0$ and $m \geq 0$ and L_1, \dots, L_m be a set of literals. Then a clause is $A_1 \vee \dots \vee A_n \vee L_1 \vee \dots \vee L_m$.

Definition 9 (Rules and programs): Let $\varphi = A_1 \vee A_2 \vee \dots \vee A_n \vee L_1 \vee \dots \vee L_m$ be a clause as defined in Definition 8. The rule notation of the clause φ is $A_1 \vee \dots \vee A_n \leftarrow L_1 \wedge \dots \wedge L_m$ where $A_1 \vee \dots \vee A_n$ is called the consequent or head and $L_1 \wedge \dots \wedge L_m$ is called the antecedent or body of the rule. For $A_1 \vee \dots \vee A_n \leftarrow \top$ and $\perp \leftarrow L_1 \wedge \dots \wedge L_m$ the notations $A_1 \vee \dots \vee A_n \leftarrow$ and $\leftarrow L_1 \wedge \dots \wedge L_m$, respectively, can be used as shorthand. A program is a finite set of rules [15].

Example 2: Let L be a signature of first-order logic. Let a, b , and c be variables. The set of rules $\{a \leftarrow b \wedge c, b \leftarrow c, c \leftarrow\}$ is a program [13].

Definition 10 (Rectification): Let φ be a formula, Q be a quantifier and x be a variable. φ is rectified if for each quantification Q of a variable x , there is neither another quantifier for x as well as no quantifier free occurrence of x in φ . A formula can be rectified by renaming variables [15].

Example 3: Let L be a signature, $R(.,.)$ and $P(.,.)$ be 2-ary function symbols in L and $\{x, y\}$ be a set of variables. Let φ be the formula:

$$\varphi = \forall x R(x, y) \wedge \exists x \exists y P(x, y)$$

In φ the variable x has both been quantified universally and existentially. y occurs both quantified and free in this formula. A possible rectification of φ would be:

$$\varphi' = \forall x R(x, w) \wedge \exists z \exists y P(z, y)$$

with z and w being variables. φ and φ' are equivalent.

Definition 11 (Ground): A term or an atom is called ground if it contains no variable [52].

Definition 12 (Substitution): Let $T = \{t_1, \dots, t_n\}$ be a finite set of terms, let $X = \{x_1, \dots, x_m\}$ be a finite set of variables and let ω be a homomorphous function mapping every variable in X to a term in T . ω is a substitution and is represented by the finite set $\{x_1 \mapsto \omega x_1, \dots, x_m \mapsto \omega x_m\}$. X is called the domain of ω and the set $\{\omega x_1, \dots, \omega x_m\}$ is called its co-domain [15].

Definition 13 (Grounding substitution): Let ω be a substitution and T a finite set of terms. ω is grounding T if for every term t in T the substitution ωt is ground. The co-domain of a grounding substitution consists only of grounded terms [13].

Definition 14 (Herbrand universe and Herbrand base): Let L be a signature. The Herbrand universe HU of L is the set of all ground L -terms. The Herbrand base HB of L is the set of all ground L -atoms. It is further assumed that L contains at least one constant c [11].

Example 4: Let L be a language. Let $a(.)$ be the only predicate in L and let 0 be the only constant in L . The Herbrand universe HU of L is $HU = \{0\}$. The Herbrand base HB of L is $HB = \{a(0), a(a(0)), a(a(a(0))), \dots\}$.

Definition 15 (Interpretation): Let L be a signature. An interpretation I over a signature L is a triple $I = (D, F, V)$. The elements of this triple are:

D is a non-empty set called the domain or universe of I .

F is a function which assigns every n -ary function f and every relation r in L an n -ary function f^I or relation r^I , respectively, such that $f^I : D^n \rightarrow D$ and $r^I : D^n \subseteq D$.

V is a variable assignment, that is, a function which maps every variable x to an element in D .

Definition 16 (Model): Let P be a program, $R = \{r_1, \dots, r_n\}$ be the set of all rules in P , and I be an interpretation of P . I is a model of P exactly if all rules of P are satisfied under I , denoted as $I \models P$, in the sense that $I \models r_1, \dots, I \models r_n$ [13].

Example 5: Let r be the rule $\top \leftarrow a \wedge b$. Let I be an interpretation with $D = \{\top, \perp\}$, $F = \{\}$, and $V = \{a \mapsto \top, b \mapsto \perp\}$. I is not a model of r since $\top \leftarrow \top \wedge \perp$ is not satisfied.

Let $I' = (D, F, V')$ be an interpretation with $V' = \{a \mapsto \top, b \mapsto \top\}$. $I' \models r$ because $\top \leftarrow \top \wedge \top$ is satisfied. I' is a model of r .

Definition 17 (Herbrand interpretation and Herbrand model): Let P be a program and let I be an interpretation of P . I is a Herbrand interpretation I_H of P if its domain is the Herbrand universe HU of P . I_H is a Herbrand model of P if $I_H \models P$ [15].

Definition 18 (Minimal Herbrand model): Let P be a program and $M = (D, F, V)$ be a Herbrand model of P . M is minimal if there exists no other Herbrand model $M' = (D', F, V)$ of P such that the domain D' is a subset of the domain D [22].

If there exists exactly one minimal Herbrand model M of P then M is called the unique minimal Herbrand model of P [48]. A negation free program always has a unique minimal Herbrand model [21].

2.2 Stable model semantics in answer set programming

A rule based program (cf. Def. 9) may contain negated atoms which allow for different semantics. One of these semantics is the stable model semantics which can be computed by performing the so-called Gelfond-Lifschitz transform. A stable model for a program may not exist but if it exists, it is identical to one of the minimal Herbrand models.

The program that arises from computing the Gelfond-Lifschitz transform has fewer rules and atoms than the original program which is why it is called the "reduct" of the original program [35], to indicate that it is a "reduced version" of the original program.

To explain how the Gelfond-Lifschitz transform works, some more definitions need to be made.

Definition 19 (Definition of a predicate): Let P be a program and let p be a predicate in P . The set of rules that contain p in their head is called the definition of p [56].

Definition 20 (Stratified program): Let P be a program. P is stratified if there is a partition $P = P_0 \cup P_1 \cup \dots \cup P_n$, with P_0, P_1, \dots, P_n being disjoint rules, such that for every predicate p the following criteria holds:

1. The definition of p is contained in exactly one of the partitions of P .
2. For $1 \leq i \leq n$, if p does not occur negated in P_i then its definition is contained in $\bigcup_{j \leq i} P_j$, otherwise its definition is contained in $\bigcup_{j < i} P_j$ [56].

A more intuitive explanation of a stratified program is that in a stratified program a predicate is only allowed to occur negated if it has been defined in an earlier partition.

Example 6: Let P be a program with the following set of rules:

$$\begin{aligned} &\{p \leftarrow \neg q \wedge \neg r, \\ &\quad q \leftarrow \neg s, \\ &\quad r \leftarrow \neg t, \\ &\quad s \leftarrow, \\ &\quad t \leftarrow\} \end{aligned}$$

A possible stratification of P is:

$$\begin{aligned} P = &\{p \leftarrow \neg q \wedge \neg r\} \cup \\ &\{q \leftarrow \neg s, r \leftarrow \neg t\} \cup \\ &\{s \leftarrow, t \leftarrow\} \end{aligned}$$

Definition 21 (Closed-world assumption): Let P be a program and a be an atom in P . A closed-world assumption means that the truth value of a is false if a is not defined in P [54]. Closed-world assumption is a concept used by relational databases. Related to the closed-world assumption is negation as failure which expresses incomplete information inside of a program or database [19].

Definition 22 (Negation as failure and weak negation): Let P be a program, a an atom in P , **not** an operator, and R be the set of all rules in P . The formula **not** a is satisfied when there is no rule in R which entails a [9].

The operator **not** represents negation as failure. Negation of the form **not** a is called *weak*

negation while a negation of the form $\neg a$ is called *strong negation*. For the strong negation $\neg a$ to be satisfied in a program P there needs to exist at least one satisfied rule in P that implies that a is false. For the weak negation **not** a to be satisfied in P there must not exist a rule that implies a to be true or false.

Example 7: Let P be a program with the rule:

$$\text{innocent}(x) \leftarrow \text{not guilty}(x)$$

Let P' be a program with the rule:

$$\text{innocent}(x) \leftarrow \neg \text{guilty}(x)$$

Both of these programs have the goal to check if a person is innocent but have two very different meanings. P considers a person x to be innocent as long as they are not proven to be guilty. P' only considers a person x to be innocent if they are proven to not be guilty [57].

Definition 23 (Gelfond-Lifschitz transformation): Let P be a stratified program, A be the set of atoms in P , R be the set of rules in P and B be a subset of the Herbrand base of P . The Gelfond-Lifschitz transformation of P with respect to B and thus the reduct Π_P of P , is produced with the following steps:

1. A grounding substitution ω (Definition 13) is applied to all rules in R , replacing all variables in A with grounded terms.
2. For every $a \in B$ all rules containing the negation $\neg a$ in their body are removed from R .
3. Every negated atom is removed from the bodies of the remaining rules in R [34].

The idea behind this transformation is to remove rules that can not be entailed.

Definition 24 (Stable model): Let P be a program and let Π_P be the reduct of P . Let M_H be the set of all Herbrand models of P . A model M with $M \models \Pi_P$ is a unique minimal Herbrand model of P if $M \in M_H$. The model M is called stable [34].

The Gelfond-Lifschitz transformation creates a negation free version of a program which has at most one minimal Herbrand model (Definition 16) that coincides with the unique minimal Herbrand model of the original program. The Gelfond-Lifschitz transformation does not change the meaning of a program.

Example 8: Let P be a program with the rules:

$$\begin{aligned} p(1,2) &\leftarrow \\ q(x) &\leftarrow p(x,y), \neg q(y) \end{aligned}$$

To find the stable model of P the Gelfond-Lifschitz transformation is applied. The first step to create the reduct Π_P is to ground the program P . This yields the following set of ground rules:

$$\begin{aligned} p(1,2) &\leftarrow \\ q(1) &\leftarrow p(1,1), \neg q(1) \\ q(1) &\leftarrow p(1,2), \neg q(2) \\ q(2) &\leftarrow p(2,1), \neg q(1) \\ q(2) &\leftarrow p(2,2), \neg q(2) \end{aligned}$$

To perform the second step a subset of the Herbrand base B of P needs to be chosen. A sensible choice of B includes the atoms $p(1,2)$ and $q(1)$. The former atom is chosen because $p(1,2)$ is a fact of the program P . The second atom is chosen as it is being entailed from the rule with the body $p(1,2), \neg q(2)$ as $q(2)$ cannot be deduced with only considering the facts in the program.

So, the second step of the Gelfond-Lifschitz transformation is performed with $B = \{p(1,2), q(1)\}$, removing all rules that contain $\neg p(1,2)$ or $\neg q(1)$. The remaining rules are:

$$\begin{aligned} p(1,2) &\leftarrow \\ q(1) &\leftarrow p(1,2), \neg q(2) \\ q(2) &\leftarrow p(2,2), \neg q(2) \end{aligned}$$

During the third step all remaining negated predicates get removed from the bodies of the rules in P . The created reduct Π_P is:

$$\begin{aligned} p(1,2) &\leftarrow \\ q(1) &\leftarrow p(1,2) \\ q(2) &\leftarrow p(2,2) \end{aligned}$$

Thus, the minimal Herbrand model of Π_P is $MH^\Pi = \{p(1,2), q(1)\}$, since the antecedent of $q(2) \leftarrow p(2,2)$ can not be entailed. Π_P is also unique, and so Π_P is also the stable model of P .

2.3 The paradigm of answer set programming

Answer set programming is a branch of logic programming which emerged through the influence of databases, non-monotonic reasoning, knowledge representation, and the stable model semantics. The use of these branches leads to answer set programming being able to provide a general-purpose approach for solving complex combinatorial, search, and optimization problems.

Databases solve queries by looking for fulfilling conditions. One of the concepts databases contributed to answer set programming is the closed-world assumption (Definition 21).

Non-monotonicity is another concept used by answer set programming. A logic is non-monotonic if a previous conclusion can be invalidated through newly gained knowledge [23]. So for a program that would mean that adding a rule can reduce the number of models a program has. Non-monotonicity is utilized by answer set programming to perform reasoning by defaults, which lets conclusions be derived through a lack of evidence that would prove them wrong.

Example 9: A standard example to depict non-monotonic reasoning has to do with flying and non-flying birds. The sentence “most birds fly” is plausible because most birds can fly even though there are a lot of exceptions like ostriches or penguins. Now looking at a specific bird, *birdie*, it would make sense to assume that *birdie* can fly since “most birds fly”. But it has never been stated what type of bird *birdie* is. Without non-monotonic reasoning it would have to be assumed that *birdie* does not fly because it has not been proven to fly for now. This could be changed however by stating that *birdie* is some kind of flying bird [53].

The stable model semantics (Chapter 2.2) provides a way for answer set programming to compute the solution for the problem at hand. The stable models of a program are solutions to the problem it expresses and are called *answer sets*. The process of computing the answer sets for a problem can be broken up into four major steps: modelling, grounding, solving, and reinterpreting [55]:

1. Modelling is the description of the problem as a set of rules. The rules of an answer set program are usually written in the programming language AnsProlog.
2. Grounding a program (Definition 13) makes it possible to determine a stable model for the program. This is done by a separate program called a grounder. The way a grounder works will be further explained in Chapter 3.2.
3. Solving the program means to compute its stable model. This process makes use of Satisfiability-Solvers or SAT-Solvers.
4. Reinterpretation is the final step and contextualizes the stable model with respect to the original problem. This step leads to the stable model being readable as a solution to the problem that the program was modelled for.

An answer set program is comprised of five common types of rules: normal rules, facts, constraints, choice rules, and weight rules and constraints [44]. Normal rules in an answer set program have the same form as rules in first-order logic (Definition 9).

Definition 25 (Fact): Let a be an atom. A rule of the form $a \leftarrow$ is called a fact and is equivalent to $a \leftarrow \top$.

Example 10: Let P be a program describing the correlation of education, age and salary of an individual x . Let the rules of P be:

$$\begin{aligned} \text{highSalary}(x) &\leftarrow \text{educated}(x), \text{employed}(x) \\ \text{employed}(x) &\leftarrow \text{ofAge}(x) \\ \text{ofAge}(x) &\leftarrow \end{aligned}$$

The rule $\text{highSalary}(x) \leftarrow \text{educated}(x), \text{employed}(x)$ says that a person x that is educated and employed has a high salary. The rule $\text{employed}(x) \leftarrow \text{ofAge}(x)$ implies that a person x who is employed must be of age. The fact $\text{ofAge}(x) \leftarrow$ means that every person x that is looked at by this program is of age.

Definition 26 (Constraint): Let $\{b_1, \dots, b_n, c_1, \dots, c_m\}$ be a set of atoms. A constraint is of the form:

$$\leftarrow b_1, \dots, b_n, \neg c_1, \dots, \neg c_m$$

This is equivalent to

$$\perp \leftarrow b_1 \wedge \dots \wedge b_n \wedge \neg c_1 \wedge \dots \wedge \neg c_m$$

A constraint implies that satisfying its body leads to a contradiction in the program [22].

Example 11: Let P be a program for colouring the vertices of a graph connected through edges. Let a constraint in P be

$$\leftarrow \text{edge}(X, Y), \text{red}(X), \text{red}(Y)$$

A model of this program would not be able to contain two connected vertices that are coloured red since that would cause the body of the constraint to be satisfied, which in turn leads to a contradiction in the program.

Definition 27 (Choice rule): Let $\{a_1, \dots, a_k, b_1, \dots, b_n, c_1, \dots, c_m\}$ be a set of atoms. A choice rule is of the form

$$\{a_1, \dots, a_k\} \leftarrow b_1, \dots, b_n, \neg c_1, \dots, \neg c_m$$

The meaning of a choice rule is that if the body of the rule is satisfied then any subset of the head, excluding the empty set, is considered satisfied [55].

The head of a choice rule is read like a disjunction of atoms, that is the head $\{a_1, \dots, a_k\}$ is equivalent to $a_1 \vee \dots \vee a_k$.

Example 12: Let P be a program which describes the colours used in a painting and let a choice rule in P be:

$$\{\text{red}(x), \text{blue}(x), \text{yellow}(x), \text{green}(x)\} \leftarrow \text{paintingColour}(x)$$

This implies that the painting x can be any combination of the colours red, blue, yellow, or green as long as the predicate $\text{paintingColour}(x)$ holds. So for example $\{\text{red}(x), \text{green}(x)\}$, $\{\text{blue}(x), \text{yellow}(x), \text{green}(x)\}$, and $\{\text{red}(x)\}$ are all valid colourings of the painting x .

Definition 28 (Weight rule): Let $\{l_1, \dots, l_n\}$ be a set of literals, $\{k, w_1, \dots, w_n\}$ be a set of integers and a an atom. A weight rule is of the form

$$a \leftarrow k \leq [l_1 = w_1, \dots, l_n = w_n]$$

A weight rule assigns a weight w_i to each of the literals l_i in the body. The head a is implied when the sum of the weights of all satisfied literals is equal to or higher than k [10].

Example 13: Let $\{\text{bread}, \text{pretzel}, \text{milk}, \text{egg}, \text{groceries}\}$ be a set of atoms and let P be a program which makes sure that groceries of at least a certain value are bought. Let P contain the weight rule:

$$\text{groceries} \leftarrow 6 \leq [\text{bread} = 2, \text{pretzel} = 1, \text{milk} = 3, \text{egg} = 4]$$

The rule is satisfied by any combination of atoms with a combined weight of 6 or more. Some models of this rule are $\{\text{bread}, \text{milk}, \text{pretzel}\}$, $\{\text{bread}, \text{egg}\}$ or $\{\text{bread}, \text{milk}, \text{pretzel}, \text{egg}\}$.

Definition 29 (Weight constraint): Let $\{l_1, \dots, l_n\}$ be a set of literals, $\{k, p, w_1, \dots, w_n\}$ be a set of integers and let a be an atom. A weight constraint is of the form

$$k \leq [l_1 = w_1, \dots, l_n = w_n] \leq p$$

The weight constraint assigns a weight w_i to each literal l_i and is satisfied when the sum of the weight of satisfied literals is between the lower bound k and the upper bound p [50].

Example 14: Let P be a program which ensures that a truck x is not too heavily loaded but is also not too lightly loaded. Let a rule in P be:

$$2 \leq [\text{barrel}(x) = 2, \text{crate}(x) = 3, \text{container}(x) = 6] \leq 6$$

The lower bound of the weight constraint being 2 ensures that x can not be empty and the upper bound 6 ensures that items with a combined weight of 7 or higher can not satisfy the rule. Some models of this rule are $\{\text{barrel}(x)\}$, $\{\text{crate}(x)\}$, $\{\text{barrel}(x), \text{crate}(x)\}$ and $\{\text{container}(x)\}$.

Definition 30 (Weight solution): Let P be a program, σ be a function which maps a model M for P to an integer and w be an integer that represents the required minimum weight of a solution. For a model to be accepted as a solution it needs to fulfil the criterion $\sigma(M) \geq w$ [16].

The rules and constraints that have been introduced above are enough to solve a wide variety of problems with answer set programming. However further research was conducted to increase the scope of problems that can be solved with answer set programming. Examples are functional dependencies, aggregates, double negations, and "Guess and Check".

Aggregates provide the ability to declaratively express the properties of sets of terms. To define the syntax of the aggregate it is necessary to define aggregate functions and aggregate elements.

Definition 31 (Aggregate function): Let T be a set of terms. An aggregate function $\#f$ maps T to a term and consists of the cardinality symbol $\#$ and a function symbol f .

Some common examples for aggregate functions are $\#count$, which determines the number of elements in the set of terms, $\#sum$, which adds up all the terms in the set of terms, $\#min$, which determines the lowest value in the set of terms, and $\#max$, which determines the highest value in the set of terms.

Definition 32 (Aggregate element): Let V be a set of variables $\{v_1, \dots, v_m\}$ with $m \geq 1$ and let ϕ be a conjunction of literals over V . An aggregate element has the syntax $v_1, \dots, v_m : \phi$,

and selects those variables in V which satisfy ϕ .

Definition 33 (Aggregate): Let $E = \{e_1, \dots, e_n\}$ be a set of aggregate elements, let $\#f$ be an aggregate function, let k be a term and let Θ be a 2-ary relation that compares two terms. An aggregate is of the form $\#f(e_1, \dots, e_n)\Theta k$. The aggregate is satisfied if the relation Θ is satisfied [2].

Example 15: Let P be a program and let a constraint in P be

$$\leftarrow \text{time}(T), \# \text{count} \{A : \text{task}(A, T)\} > 1$$

The predicate $\text{time}(T)$ in the body of the constraint is used to represent a time interval T . The aggregate $\# \text{count} \{A : \text{task}(A, T)\} > 1$ is satisfied if there is more than one term A that satisfies $\text{task}(A, T)$. The constraint causes a contradiction in P if the aggregate is satisfied. That means there can not be multiple $\text{task}(A, T)$ for a unique time interval T .

CHAPTER 3

Answer set programming with AnsProlog

The language AnsProlog or A-Prolog is the standard language for creating answer set programs. AnsProlog is based on the declarative language Prolog. While AnsProlog possesses many functionalities and the syntax of Prolog, it also allows for choice rules, aggregates, and constraints.

3.1 Syntax and semantics

AnsProlog is a language based on facts and rules, which evaluates all stable models for a given program and presents those as answers.

A fact in AnsProlog has the following syntax:

```
p(x1, ..., xn) .
```

where n denotes the arity of the predicate $p(x_1, \dots, x_n)$. Facts are considered to be true in the program. The dot "." marks the end of facts and rules.

A rule in AnsProlog has the following syntax:

```
a :- b1, ..., bn .
```

where a, b_1, \dots, b_n are predicates. The rule corresponds to the logic formula $a \leftarrow b_1 \wedge \dots \wedge b_n$. A comma "," is seen as a conjunction (" \wedge ") and the connector ":-" corresponds to an implication.

An example in the form of an AnsProlog program, which also happens to be a Prolog program, illustrates these basic concepts.

Example 16: Code Example 3.1 computes the paths in a directed graph that consists of nodes and connecting edges.

```
node(1) .    edge(1,2) .
node(2) .    edge(1,4) .
node(3) .    edge(2,4) .
node(4) .    edge(3,1) .

path(Id,Id) :- node(Id) .
path(From,To) :- edge(From,Stop), path(Stop,To) .
```

Code Example 3.1: Paths in a directed graph

The four facts with the `node` predicate number the nodes in the graph. The facts with the `edge` predicate set up edges between nodes, stating at which node the edge originates and at which node it ends.

The rule "`path(Id,Id) :- node(Id) .`" sets up a reflexive property, so that every node has a path from and to itself.

The rule "`path(From,To) :- edge(From,Stop), path(Stop,To) .`" declares all other paths between nodes.

Executing this program leads to the following solution:

```
{path(1,1); path(2,2); path(3,3); path(4,4); path(1,2);
path(1,4); path(2,4); path(3,1); path(3,2); path(3,4)}
```

Duplicate paths do not occur in this solution. The solution includes all paths that are available between nodes inside of the directed graph [45].

Next, the syntax and semantics of constraints in AnsProlog needs to be discussed. A constraint in AnsProlog has the following syntax:

```
:- a1, ..., an .
```

where a_1, \dots, a_n are predicates. The constraint corresponds to the logic formula $\leftarrow a_1 \vee \dots \vee a_n$.

Example 17: Code Example 3.2 is an AnsProlog program that computes the colouring of the nodes of a directed graph which are connected by edges, such that connected nodes do not have the same colour.

```
node(1).    red(X).
node(2).    blue(X).
node(3).    green(X).

edge(1,2).  edge(2,3).
edge(1,3).  edge(3,1).
edge(2,1).  edge(3,2).

:- edge(X,Y), red(X), red(Y).
:- edge(X,Y), blue(X), blue(Y).
:- edge(X,Y), green(X), green(Y).
```

Code Example 3.2: Coloured graph nodes

The predicates for nodes and edges are used as in Example 16. The facts `red`, `blue` and `green` define the three colours available for colouring. The constraints at the end of the program cause a contradiction if two connected nodes are of the same colour, and as such filter out all colour combinations that do not fulfil the prerequisite of the posed problem. Executing this program presents the following set of solutions:

```
{{red(1), blue(2), green(3)}; {red(2), blue(3), green(1)};
{red(3), blue(1), green(2)}}
```

This example shows how constraints are used in AnsProlog and how they reduce the number of stable models for a given program by filtering out solutions that do not fulfil their condition.

Having discussed facts and rules, the syntax of negations in AnsProlog will be explained next. AnsProlog supports both kinds of negation as described in Definition 5 and Definition 22. The classical form of negation is expressed in AnsProlog as:

```
-a
```

whereas negation as failure is expressed as:

```
not a
```

while a represents a predicate in both code examples.

Example 18: Code Example 3.3 shows the use of both kinds of negations.

```
-b.
c :- not a, -b.
```

Code Example 3.3: Negations

where a , b , and c are predicates. The predicate b is in fact false due to the first line. The predicate a , on the other hand, has no rule or fact proving it to be false or true, which satisfies the weak negation "`not a`". AnsProlog computes $\{-b, c\}$ as an answer to this

program.

Answer set programming can also deal with choices expressed by choice rules. The syntax for choice rules (Definition 26) in AnsProlog is:

```
{a}.
```

and means that *a* can either be true or false in a program. When choice rules are used in an AnsProlog program, the number of solutions increases combinatorially [33].

Example 19: Code Example 3.4 shows a choice rule.

```
{p}.
```

Code Example 3.4: Choice rule

The two solutions of Code Example 3.4 are either that *p* is false or that *p* is true, which is expressed as the two sets {} and {*p*} in AnsProlog.

Code Example 3.5 shows Code Example 3.4 with one additional choice rule containing the predicate *q*.

```
{p}.
{q}.
```

Code Example 3.5: Added choice rule

The solutions for Code Example 3.5 are {}, {*p*}, {*q*}, and {*p*, *q*}. This shows how choice rules exponentially increase the amount of possible solutions.

Choice rules can also be declared in an abbreviated way bringing them further in line with their definition in Chapter 2.3. The choice rule of the form:

```
{p;q}.
```

is equivalent to the following two choice rules:

```
{p}.
{q}.
```

Additionally, choice rules can contain a rule body as described in Definition 27, that is a rule body which contains multiple atoms.

Aggregates and aggregate functions are also implemented in AnsProlog. The form of the aggregate is identical to the way it was defined in Definition 33. The constraint from Example 15 can be used in AnsProlog as follows:

```
:- time(T), #count{A:task(A,T)} > 1.
```

Code Example 3.6: Timed aggregate

where *time*(*T*) is a predicate that represents time intervals and *task*(*A*, *T*) is a predicate that represents a task *A* that can be performed during the time *T*.

The template aggregate functions provided by AnsProlog are #count{*X*}, #sum{*X*}, #max{*X*}, #min{*X*}, #max{*X*, *Y*}, and #min{*X*, *Y*} [2].

Some of the constructs in AnsProlog offer syntactical shortcuts to describe larger amounts of rules or facts in a single line of code. AnsProlog provides the ability to use ranges:

```
a(1..n).
```

Code Example 3.7: Abbreviations

which is an abbreviation for the following enumeration of facts:

```
a(1).
...
...
...
a(n).
```

Weight constraints and rules have two variations in AnsProlog: Cardinality constraints and weight constraints [14]. The syntax of cardinality constraints is:

```
a{p1(X), ..., pn(X)}b.
```

where a and b are numbers defining an upper and lower bound and $p1(X), \dots, pn(X)$ are predicates. This cardinality constraint is satisfied if the number of satisfied predicates it contains is between a and b .

The second variation of a weight constraint has the following syntax:

```
a{p1(X)=w1, ..., pn(X)=wn}b.
```

where a and b are numbers defining a lower and an upper bound, $p1(X), \dots, pn(X)$ are predicates, and $w1, \dots, wn$ are floating point numbers. This type of weight constraint assigns a weight to every predicate. The weight constraint is satisfied, if the sum of the weights of its satisfied predicates is between a and b [38].

Example 20: The Code Example 3.8 shows a cardinality constraint that makes sure that a node has exactly one colour.

```
node(1). node(2). node(3).
1{coloured(X,blue), coloured(X,green), coloured(X,red)}1 :- node(X).
```

Code Example 3.8: Cardinality constraint

The first line in Code Example 3.8 numbers the nodes as in Code Example 3.1. In the second line a cardinality constraint serves as the head of a rule. The rule makes sure that every node is assigned exactly one colour. This condition is achieved by setting the lower and upper bound of the cardinality constraint to 1, so that for every node exactly one predicate `coloured` holds for a particular solution.

Example 21: Code Example 3.9 shows an AnsProlog program which ensures that a truck X is not too heavily loaded.

```
item(barrel). item(crate). item(container).
loadTruck(X) :- 1{on(X,barrel)=2, on(X,crate)=3, on(X,container)=6}8.
```

Code Example 3.9: Weight rule

The three facts consisting of the `item` predicate define the items that can be loaded on a truck. The weight rule limits how many of the items can be put on the truck to satisfy `loadTruck(X)`. The sum of the weights of the satisfied items is only allowed to be between 1 or 8. For a truck 1 the solutions are:

<pre>{{on(1,barrel),loadTruck(1)}; {on(1,barrel),on(1,crate), loadTruck(1)}; {on(1,barrel),on(1,container),loadTruck(1)}}</pre>

3.2 Grounding

An integral step during answer set programming is grounding. Grounding is required for computing the stable model of a program.

Example 21: Code Example 3.10 shows a program that is not yet grounded.

```
entity(1).      entity(2).
tupel(X1,...,Xn) :- entity(X1),...,entity(Xn).
```

Code Example 3.10: Grounding size

This program consists of two facts and a single rule. Grounding this program binds all variables with constants by creating rules for every possible combination. Every `entity(Xi)` in the body of the rule needs to be substituted with `entity(1)` or `entity(2)`, so that every possible interpretation is represented.

The grounded version of Code Example 3.10 contains 2^n ground rules, corresponding to the number of n -tuples over a set of two elements [39]. The size of the grounded program shows that even relatively simple and short programs might require large amounts of space for grounding.

The version of a program that gets created through grounding (Definition 13) is called an *instantiation* of the program.

Applications called grounders deal with the instantiation of programs. The purpose of grounders is not only to ground the program but also to simplify and reduce the size of instantiations.

Example 22: Code Example 3.11 shows a program that is about to be instantiated.

```
a(1,2).
b(X),c(Y) :- a(X,Y).
```

Code Example 3.11: Instantiating

The instantiation of the last line in Code Example 3.11 leads to the following four rules:

```
b(1),c(2) :- a(1,2).
b(1),c(1) :- a(1,1).
b(2),c(1) :- a(2,1).
b(2),c(2) :- a(2,2).
```

Only the first of these instantiations can be satisfied since no facts justify the other rules. The other instantiations are not relevant for solving the problem, and can be omitted while still preserving the meaning of the program.

Some notable grounders are *lparse* [47], *GrinGo* [32], and the *DLV-2* system [1]. Grounders use different approaches for grounding but the main idea is as follows:

1. A set of constants and ground predicates gets defined as an extension E . In a program the extension is the set of all ground predicates and constants.
2. The first non-ground rule r in a program is chosen for grounding.
3. Variables in r are matched with elements in E and substituted by them.
4. This substitution is reiterated over all variables in r until a ground rule is created.

5. The grounder performs a backtracking step, if a ground rule is created or there exist no more matches for an unsubstituted variable in r . The backtracking step reverts the last variable substitution and checks for an additional match in E .
6. The instantiation of r is complete when there are no more possible backtracking steps.

An example will illustrate how a grounder creates the instantiation of a single rule while using a given extension.

Example 23: Let the rule r be:

$$a(X), b(Y) \text{ :- } p(X, Z), q(Z, Y).$$

Code Example 3.12: Instantiating of a rule

and let $E = \{p(1, 2), q(2, 1), q(2, 3)\}$ be the extension of r . The grounder creates the instantiation from left to right in the body of the rule, trying to match elements of E with the predicates in the body of the rule r .

Starting with $p(X, Z)$ the grounder checks for matching predicates in E . The first match is $p(1, 2)$. The grounder substitutes 1 for X and 2 for Z and moves on to match $q(Z, Y)$. For $q(Z, Y)$ the variable Z is already bound to 2, so the grounder tries to match the predicate $q(2, Y)$ in E . It finds $q(2, 1)$ and substitutes 1 for Y . With that, the entire body is ground and the rule:

$$a(1), b(1) \text{ :- } p(1, 2), q(2, 1).$$

is created. The grounder performs a backtracking step after a rule has been created and returns to the previous matching process for $q(2, Y)$. It finds one more match in $q(2, 3)$ and creates the grounded rule:

$$a(1), b(3) \text{ :- } p(1, 2), q(2, 3).$$

The grounder again performs a backtracking step after the creation of a grounded rule. Now the grounder cannot find any more matches for $q(2, Y)$ so it performs an additional backtracking step, and checks for more matches for $p(X, Z)$. The grounding process ends here because there are no more matches left for $p(X, Z)$ and there are no more possible backtracking steps. The grounded rules generated through this process are:

$$\begin{aligned} a(1), b(1) &\text{ :- } p(1, 2), q(2, 1). \\ a(1), b(3) &\text{ :- } p(1, 2), q(2, 3). \end{aligned}$$

To ground a program P , the process of grounding a rule gets repeated from top to bottom for all rules in P . At the start of the instantiation, the extension of the rules is the set of all facts in P . New predicates that are proven through the grounding of previous rules are added to the extension.

Example 24: Code Example 3.13 shows a program that needs to be grounded and is adapted from [39].

$$\begin{aligned} &p(1, 2). \quad q(2, 1). \quad q(2, 3). \\ a(X), b(Y) &\text{ :- } p(X, Z), q(Z, Y). \\ c(X) &\text{ :- } a(X). \end{aligned}$$

Code Example 3.13: Instantiating of a program

The initial extension E of P is $\{p(1, 2), q(2, 1), q(2, 3)\}$. The first rule and the extension are the same as in Example 23, as such the first generated grounded rules of the instantiation are also :

```
a(1), b(1) :- p(1,2), q(2,1).
a(1), b(3) :- p(1,2), q(2,3).
```

This instantiation entails the predicates $a(1)$, $b(1)$, and $b(3)$, which are added to the extension E . The grounder moves on to the rule $c(X) :- a(X)$ and checks for matches for $a(X)$ in E . It finds the atom $a(1)$ and substitutes 1 for X in the rule $c(X) :- a(X)$, thus creating the rule:

```
c(1) :- a(1).
```

The grounder performs a backtracking step to check for more matches for $a(x)$. There are no more matches left, so the grounder terminates. The grounded program is:

```
p(1,2). q(2,1). q(2,3).
a(1), b(1) :- p(1,2), q(2,1).
a(1), b(3) :- p(1,2), q(2,3).
c(1) :- a(1).
```

At the end of the grounding step, the extension is $\{p(1,2), q(2,1), q(2,3), a(1), b(1), b(3), c(1)\}$.

Grounding a recursive rule generates grounded predicates that are required for the creation of further grounded instances of the rule. In a recursive rule, the grounding process must be iterated until a fixpoint is reached. After every iteration, the extension of the rule becomes the predicates that were satisfied during the previous iteration, as well as all the constants and predicates in the starting extension. The grounding process ends if no new predicates can be added to the extension.

Example 25: Code Example 3.14 shows an AnsProlog program that computes paths in a directed graph.

```
edge(1,2). edge(2,3). edge(3,4).
path(From,To) :- edge(From,To).
path(From,To) :- edge(From,Stop), path(Stop,To).
```

Code Example 3.14: Recursive instantiation

The nodes of the graph are not explicitly stated but implicitly given by their connected edges. $E = \{edge(1,2), edge(2,3), edge(3,4)\}$ is the extension of the program shown in Code Example 3.14. The first rule defines all paths that directly connect two nodes. The second rule uses recursion to create all paths which connect more than two nodes.

The rules that are created when grounding the first rule are :

```
path(1,2) :- edge(1,2).
path(2,3) :- edge(2,3).
path(3,4) :- edge(3,4).
```

All three of the implied $path(X,Y)$ predicates are added to E . For the first iteration of the second rule, the same grounding process gets performed as in Example 26 and creates the following grounded rules:

```
path(1,3) :- edge(1,2), path(2,3).
path(2,4) :- edge(2,3), path(3,4).
```


In the next iteration of the grounding process, the extension of the rule is modified such that $E = \{\text{path}(1,2), \text{path}(1,3), \text{path}(2,3), \text{path}(2,4), \text{path}(3,4), \text{edge}(1,2), \text{edge}(2,3), \text{edge}(3,4)\}$. Performing the grounding process using the new extension yields the grounded rule:

```
path(1,4) :- edge(1,2),path(2,4).
```

In the next iteration $E = \{\text{path}(1,2), \text{path}(1,3), \text{path}(1,4), \text{path}(2,3), \text{path}(2,4), \text{path}(3,4), \text{edge}(1,2), \text{edge}(2,3), \text{edge}(3,4)\}$. Performing the grounding process using the new extension yields no new rules, and consequently, a fixed point was reached. The final instantiation of the program is:

```
path(1,2) :- edge(1,2).
path(2,3) :- edge(2,3).
path(3,4) :- edge(3,4).

path(1,3) :- edge(1,2),path(2,3).
path(2,4) :- edge(2,3),path(3,4).
path(1,4) :- edge(1,2),path(2,4).
```

Code Example 3.15: Grounded recursive rule

The syntax and methods that have been presented in the current and preceding chapters provide only a very basic overview over AnsProlog and grounding. However, AnsProlog also supports more sophisticated features, such as GZ-aggregates [3], inductive logic-programming [20], and consistency restoring rules [6].

The following chapter will highlight some implementations of answer set programming in order to not only show the concepts of answer set programming but also provide a more hands-on perspective on this topic.

Practical Aspects of Answer Set Programming

The implementation of answer set programming requires applications that can evaluate and solve the programs written in AnsProlog. A variety of such applications have been developed since the introduction of answer set programming; some examples include *Smodels*, *Cmodels*, *ASSAT*, *DLV*, and *Clasp*. *Smodels* and *Clasp* are good examples to show the evolution of answer set programming and thus will be looked at in more detail.

Smodels was introduced in 1995 and is one of the oldest answer set solvers [59]. The *Smodels* implementation requires an input program to be grounded, and as such needs to be paired with a grounder such as *lparse*. *Smodels* computes the solution of a program by utilizing a backtracking algorithm that searches for the stable models of the input program [47].

Clasp was introduced in 2007 and offers a conflict-driven approach to answer set solving. It is part of the application bundle *Clingo*, which includes *Clasp* as an answer set solver and *Gringo* as a grounder. *Clasp* uses the *nogoods* learning algorithm to find a solution for a program [28].

The difference between *Smodels* and *Clasp* is fundamental: *Smodels* is able to compute the solution for programs containing normal and choice rules, constraints, and weight constraints, while *Clasp* is able to additionally allow for functions, aggregates, and strong negations.

The performance of *Clasp* and *Smodels* also differs: *Clasp* uses a more effective approach to answer set solving. *Clasp* outperforms *Smodels* in both space and time requirements, as multiple benchmarks indicate [29, 18].

Both of these answer set solvers and their approaches will be described to highlight how the implementation of answer set solving has evolved. This comparison will be supplemented by an actual application of answer set programming, in the form of a space shuttle decision system.

4.1 Smodels and lparse

Smodels is a C++ application that was developed to implement the stable model semantics in a way to be used in realistic applications. The development of *Smodels* was closely linked to the development of the grounder *parse*, which would later become *lparse* [49].

The algorithm used by *Smodels* is not able to deal with non-grounded programs, and as such uses *lparse*. The *lparse* application does not only ground a program, but also removes superfluous rules from the instantiation, as described in Chapter 3.2. It should be stated that both *lparse* and *Smodels* work as filters which means that they read from standard input and their results are written to standard output.

The input for *lparse* is a logic program written in a Prolog style syntax which is also allowed to contain constraints, built-in functions, choice rules, and cardinality and weight constraints. Additionally, only *domain-restricted* programs are accepted by *lparse*. What domain-restricted programs are, is best explained through an example.

Example 26: Code Example 4.1 shows an AnsProlog program that determines the family tree of some individuals called *jack*, *jill*, and *joan*.

```

ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y), person(X).
ancestor(X,Y) :- parent(X,Y).
son(X,Y) :- parent(Y,X), male(X).
daughter(X,Y) :- parent(Y,X), female(X).
person(X) :- male(X).
person(X) :- female(X).

parent(jack, jill). parent(joan, jack).
male(jack). female(jill). female(joan).

```

Code Example 4.1: Domain-restricted program

The predicates of Code Example 4.1 can be divided into two classes, *domain predicates* and *non-domain predicates*. Domain predicates are all the predicates that are defined non-recursively, while all other defined predicates are non-domain predicates. For *P* that means that its only non-domain predicate is *ancestor(X, Y)*, since the first rule defines it recursively.

A rule is called *domain-restricted* if every variable that appears in the head of a rule also appears in an unnegated domain predicate in its body. A program is domain-restricted if every rule it contains is domain-restricted.

In the first rule all three variables *X*, *Y*, and *Z* appear in the *parent* and *person* predicates. The first rule is domain-restricted, since the variables *X*, *Y*, and *Z* appear in the two domain predicates *parent(Z, Y)* and *person(X)*. This criterion holds for all the other rules as well, which means that the program is domain-restricted [47].

The reason for allowing only domain-restricted programs in *lparse* is that it is necessary to know which rules allow for effective grounding. If, for example, a program was to be grounded that contains the rule:

```

a(X) :- not b(X,Y).

```

there needs to be a grounded version of the rule for every possible substitution for *Y*. In this sense, the domain predicates are used to “restrict the domain” of the variables in a program [62].

The first step *lparse* takes for a domain-restricted logic program is to process its rules by

performing the matching process described in Chapter 3.2. This creates the grounded version of the program. *lparse* then translates all rules in the program into primitive rules, that is rules which only consist of predicates, conjunctions, disjunctions, negations, and implications. This translation step is performed to make it possible for *Smodels* to compute the stable model of more complicated constructs like weight constraints.

The grounded primitive program is then once again translated, this time into a numerical representation, which is the output of *lparse*. This numerical representation, in turn, is the input for *Smodels*.

The algorithm employed by *Smodels* to determine the stable model of its input program is based on the Davis-Putnam procedure. The Davis-Putnam procedure proves a formula ϕ by showing that a ground instance of $\neg\phi$ is unsatisfiable [66].

The *Smodels* application roughly performs these three steps: *expand*, *conflict*, and *look-ahead*. The expand step is the assignment of truth values to unassigned atoms in the program. The conflict step checks for contradictions in the program which would lead to backtracking. The look-ahead step uses the Davis-Putnam procedure to check if an atom is satisfied in the stable model.

If a stable model has been found, then *Smodels* saves that model and performs a backtracking step to look for further stable models. In case no stable model could be derived, *Smodels* "looks ahead" by checking if there is a predicate in the program which is weakly negated, and checks whether a solution is possible with the opposite truth value. *Smodels* either adds the predicate from the look-ahead step to the potential stable model if it does not cause a contradiction, or omits it if a negation was found.

All the steps are then repeated iteratively by *Smodels* with the predicates that were derived to be contained in the stable model. The process terminates once there are no more predicates left that could potentially be assigned a truth value.

Example 27: Code Example 4.2 shows an AnsProlog program that computes the colouring of nodes in a graph under the condition that connected nodes can not be of the same colour.

```
color(1..2).      node(1..2).
edge(1,2).
1 {nodeColor(N, C):color(C)} 1 :- node(N).
:- nodeColor(X, C), nodeColor(Y,C), edge(X,Y), color(C).
```

Code Example 4.2: *Smodels* algorithm

The instantiation of Code Example 4.2 is:

```
color(1..2).      node(1..2).
edge(1,2).
1 {nodeColor(1, 1),nodeColor(1, 2)} 1 :- node(1).
1 {nodeColor(2, 1),nodeColor(2, 2)} 1 :- node(2).
:- nodeColor(1, 1), nodeColor(2,1), edge(1,2), color(1).
:- nodeColor(1, 2), nodeColor(2,2), edge(1,2), color(2).
```

Executing this program with *Smodels* yield two stable models:

```
{color(1..2), node(1..2), edge(1,2), nodeColor(1,1), nodeColor(2,2)}
```

and

```
{color(1..2), node(1..2), edge(1,2), nodeColor(1,2), nodeColor(2,1)}
```

Smodels is being executed together with *lparse* by calling the following command in a terminal window:

```
% lparse input_program | smodels
```

Additional command line options that can be given to *lparse* are, for example, `-d X, ..., Y`, which sets the used domain for the grounding process to `X, ..., Y`, and `-c const=n`, which overrides the value of a constant `const` in the input program with the value `n`.

Command line options that can be given to *Smodels* are, for example, `nolookahead`, which skips the look-ahead step of the algorithm, and a number behind *Smodels* indicates how many stable models should be computed.

Example 28: Let `colour.lp` be the graph colouring program from Example 31. A command line that uses the facts in the program as a domain, that does not use the look-ahead step, and computes one stable model is:

```
% lparse -d facts colour.lp | smodels 1 -nolookahead
```

The resulting output for running this command line is:

```
Answer: 1
Stable Model: nodeColor(1,1),nodeColor(2,2)
```

This shows that *Smodels* omits the domain used in the program from the presented stable model.

Looking at the performance and requirements of *Smodels* and *lparse* shows that they offer an effective solution for solving small answer set programs. The complexity of finding the stable model of a grounded program with all the constructs that are accepted by *Smodels* has been proven to be part of **NP** [50]. Additionally, the space required for the grounding process of *lparse* grows in a linear way [47].

Since its complexity and spatial requirements are so low, *Smodels* is a solid choice for modelling and solving combinatorial problems, the complexity of which mostly lies in **NP-hard**, and which require large amounts of space for computing all possible configurations.

Nowadays, *Smodels* is being outperformed by many of the newer solvers. *Smodels* is, however, often still used as a benchmark for other solvers because of its historical importance as one of the first answer set programming applications [18].

4.2 Clasp and GrinGo

The answer set solver *Clasp* and the grounder *GrinGo* were introduced in 2007 and are, at the time of writing, being distributed as a single application called *Clingo* [26]. *Clasp* offers a completely new approach to answer set solving, while *GrinGo* improves upon previously introduced grounders, namely *dlv* and *lparsc*.

The process and order of solving a program with the help of *GrinGo* and *Clasp* is the same as with *lparsc* and *Smodels*. The program is input into *GrinGo*, which generates a primitive grounded program so that *Clasp* can compute its stable models. The computed stable models are then presented in text form.

GrinGo and *Clasp* are also invoked by means of a command line with a nearly identical syntax to the one shown in Example 31. The syntax of a basic command line for *GrinGo* and *Clasp* is:

```
% gringo input_program | clasp
```

Many command line options which work for *Smodels* and *lparsc* also work with *GrinGo* and *Clasp*, for example setting the domain with `-d` and adding a number after the *Clasp* command name to define how many stable models should be computed.

GrinGo and *Clasp* seem rather similar to *Smodels* and *lparsc* from the outside. Their differences, however, will become apparent on a closer look.

The grounding process of *GrinGo* can be divided into four steps. The first step is *parsing*, where *GrinGo* checks if the syntax of the input program is correct. In that case, an internal representation of the program is created, and *GrinGo* moves on to the next step.

The second step is *checking*, in which *GrinGo* verifies that the input program has a finite ground instantiation that does not change the meaning of the original program. The checking step uses this analysis of the program to determine and schedule which predicates are going to be grounded first.

The third step is *instantiation*, in which the scheduled predicates are grounded to create the instantiation. The method used to effectively ground the rules is based on the *back-jumping* algorithm used by *dlv*.

The fourth and final step is *evaluation*, in which the grounded rules from the previous step are examined to identify all newly derivable predicates. Those new predicates are then scheduled to be grounded and used to repeat the instantiation step. Additionally, the simplification of the grounded program, as has been shown in Chapter 3.2, is also performed during this step. If there are no new derivable predicates, the grounded program is being output [32].

The algorithm that is utilized by *GrinGo* during the instantiation step is an enhanced version of the *back-jumping* algorithm. The idea behind this algorithm is to distinguish between the predicates that contain *relevant* variables, and the ones that contain *irrelevant* variables. A variable is considered irrelevant in a rule if it is contained in a predicate for which all the truth values of each of its grounded instances are known. The algorithm then avoids revisiting predicates with irrelevant variables for new substitutions.

GrinGo also splits rules into two parts: one holding only relevant variables, the other holding irrelevant variables. The instance containing irrelevant variables contains also relevant variables if they are dependencies of irrelevant variables.

It is possible for *GrinGo* to skip the generation of redundant grounded rules by skipping the repeated substitution of irrelevant variables. These substitutions always lead to the same result for the rule [29]. An example should illustrate how this algorithm is applied and

what it achieves.

Example 29: Code Example 4.3 shows a program that is about to be grounded and consists of the following set of rules:

```
x(1, 1..4).
y(A) :- x(A, B), not z(A).
```

Code Example 4.3: Relevant and irrelevant variables

The variable A is relevant in the second line because it appears in $\text{not } z(A)$, which has a grounded instance with an unknown truth value. The variable B is irrelevant in the second rule because it only appears in $x(A, B)$ for which all ground instances are defined through the fact in the first line.

The predicate x is split into $x(A, *)$ and $x(A, B)$ by *GrinGo* to represent the relevant and irrelevant variables. When *GrinGo* grounds the first rule, it looks for matches for $x(A, *)$ first, finding $x(1, *)$. This matching step is followed up by another for $x(1, B)$, finding $x(1, 1)$ and creating the grounded rule:

```
y(1) :- x(1, 1), not z(1).
```

GrinGo performs a backtracking step after the creation of a grounded rule, but because B is an irrelevant variable, it performs another backtracking step and checks for more matches for $x(A, *)$. *GrinGo* finds no more matches and terminates after the creation of the one grounded rule.

This example shows how *GrinGo* avoids the creation of additional rules, which all effectively have the same meaning and effect on the program.

The grounding process of *GrinGo* offers faster and more consistent run times compared to *lparse*, since it is able to skip a large amount of irrelevant variable substitutions.

Moving on to *Clasp*, it is important to first explain what *nogoods* is since it is a central part of how *Clasp* solves a program.

Definition 34 (Nogoods): Let $a_1, \dots, a_m, a_n, \dots, a_l, r$ be atoms, let \mathbf{T} and \mathbf{F} be predicates that evaluate to the truth values true and false, respectively, and let v_r be the rule:

```
r :- a_1, \dots, a_m, \neg a_n, \dots, \neg a_l.
```

The rule can be represented by two implications, the first one being

$$r \rightarrow a_1 \wedge \dots \wedge a_m \wedge \neg a_n \wedge \dots \wedge \neg a_l$$

which is equivalent to the conjunction

$$(\neg r \vee a_1) \wedge \dots \wedge (\neg r \vee a_m) \wedge (\neg r \vee \neg a_n) \wedge \dots \wedge (\neg r \vee \neg a_l)$$

The second implication is

$$a_1 \wedge \dots \wedge a_m \wedge \neg a_n \wedge \dots \wedge \neg a_l \rightarrow r$$

The conjunction of the first implications is used to derive a set of truth value assignments for which v_r is not satisfied. These sets of truth value assignments are nogoods [25].

$$\delta(v_r) = \{\{\mathbf{T}v_r, \mathbf{F}a_1\}, \dots, \{\mathbf{T}v_r, \mathbf{F}a_m\}, \{\mathbf{T}v_r, \mathbf{T}a_n\}, \dots, \{\mathbf{T}v_r, \mathbf{T}a_l\}\}$$

The second implication gives rise to the nogood

$$\Delta(v_r) = \{\mathbf{F}v_r, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_n, \dots, \mathbf{F}a_l\}$$

A short example will show how these nogoods are utilized during program solving.

Example 30: Code Example 4.4 shows a small AnsProlog program which is adapted from [25].

```
a :- b.
a :- -c.
```

Code Example 4.4: Nogoods

The resulting nogoods of the rules of Code Example 4.4 are

$$\{\mathbf{T}a, \mathbf{F}b, \mathbf{F}-c\}, \{\{\mathbf{F}a, \mathbf{T}b\}, \{\mathbf{F}a, \mathbf{T}-c\}\}$$

These nogoods are used to determine the resulting truth value assignments of variables when a different variable gets assigned a truth value. The resulting truth value assignments for $\mathbf{F}a$ through the nogood $\{\{\mathbf{F}a, \mathbf{T}b\}, \{\mathbf{F}a, \mathbf{T}-c\}\}$ are $\mathbf{F}b$ and $\mathbf{F}-c$, because fulfilling a nogood would lead to a contradiction in the program. For solving a program, these resulting truth value assignments mean that if a is proven to be false in the program then b and $-c$ must also be false. These nogoods are used during the incremental construction of the stable model to effectively assign truth values to multiple variables at once while avoiding contradictions in the program.

The system architecture of *Clasp* consists of three distinct parts. The first part is the *parser* that takes *lp* programs as input and translates them into an internal representation.

The second part is the *program builder*, which generates the nogoods representation of the input program. The program builder additionally creates an atom-body-dependency graph that represents the variables which need to be assigned truth values to satisfy the body of rules and, in turn, satisfy their heads. The non-trivial parts of this graph, so the parts in which different variable assignments lead to different outcomes, are then input into the third part, alongside the grounded program and the set of nogoods.

The third and final part of *Clasp* is the *solver*. The solver holds two databases; one for the originally generated or static nogoods, and another for nogoods that are generated from new conflicts or loops inside of the program. These newly generated nogoods are called recorded nogoods, and have an added activity counter. This activity counter allows *Clasp* to delete nogoods that are rarely important during the solving step, and as such are not required to be saved in the active database. The solver additionally sets up a watch list for variables that are part of nogoods with more than three elements. This watch list is used to quickly and easily update longer nogoods once a variable is assigned a truth value [28].

The solver starts the solving process once the data structures have been generated. The algorithm behind *Clasp* starts by assigning truth values to unassigned variables, either through propagation or a decision heuristic.

Propagation in *Clasp* means applying the *unit clause rule* to its nogoods. The unit clause rule states that if a clause is a unit clause, meaning a rule with only one unassigned literal in its body, the literal must be assigned the truth value that satisfies the rule [27]. Applying propagation to nogoods results in the derivation of additional atoms. These derived atoms are assigned a pointer to the nogoods they were derived from to make future conflict solving and backtracking easier and faster. The body-dependency graph helps to quickly apply

the propagation and compute the resulting new variable assignments.

Conflict resolution is engaged when propagation causes a conflict in the program, meaning cases in which a new nogood would be propagated which causes another nogood to be fulfilled. The conflict resolution determines and records the nogood which causes the conflict, and performs backtracking to the propagation step in which the conflict nogood was originally derived.

The decision heuristics that is applied if propagation is not possible is one in which the solver utilizes its recorded or learned conflict nogoods. The heuristics assigns values to the yet unassigned literals in the program, with the values being determined by how often the literal appears in yet unsolved nogoods as well as in conflict nogoods. The heuristic values are re-evaluated every time a new nogood is derived. The solver then assigns a truth value to the literal with the highest heuristic value and proceeds like it does during propagation. There are multiple other heuristics that are offered by *Clasp* and that can be applied through an additional command in the command line. These additional heuristics include *VSIDS* [46], *BerkMin* [37], and *VMTF* [41]. It should be added that *Clasp* offers not to use learned nogoods for its decision heuristics, and instead use a look-ahead approach to check for literals that might cause conflicts later on.

The solver creates a stable model once every literal in the program has an assigned truth value, and none of the nogoods is fulfilled. The solver then performs backtracking steps to check for additional stable models, if the initial invoke command demands more than one.

Clasp utilizes additional methods to increase its performance. These methods include *restart policies*, in which the solving process gets restarted with a different starting strategy if too many conflicts occur, and *nogood deletion*, in which unused and underutilized nogoods get deleted from the record to both save memory space and reduce the number of nogoods that have to be checked.

The above description of *Clasp* serves as an overview and is not comprehensive. It should, however, convey the idea and outline of how *Clasp* solves programs by looking for conflicts through nogoods, and constructs solutions by finding ways to avoid these conflicts.

Clasp is one of the most efficient and fastest answer set solvers at the time of writing. *Clasp*, and modified versions of it, won in most categories of the fifth answer set programming competition in 2016 [17] and are still performing well in the benchmarks set by the seventh answer set programming competition [31], which include optimization and decision problems, as well as query-answering.

When comparing *Clasp* to *Smodels*, it is evident that the former is not only more robust, through its restart options and conflict resolution, but also requires less space and time for solving a program because of its use of pointers, nogood deletion, and more effective back-jumping methods [28].

4.3 A decision support system for a Space Shuttle - An example

A well known application of answer set programming is a decision support system for space shuttles. This system is one of the first practical applications of answer set programming, and is outlined in an article from 2001 [51], which, in turn, is based on multiple other works. These other works developed a concept for a core module M_0 , which is used for verifying and finding plans for the operations performed by the reaction control system (RCS) of a space shuttle [7, 36, 65].

The RCS of a space shuttle is controlled by the astronauts during manoeuvres in orbit through flipping switches that control valves or power specific circuitry. There are pre-scripted plans that the astronauts can follow to achieve specific goals. However, issues start arising if there is any kind of failure in the system, for example a faulty switch. The number of plans that would be required to deal with all of these eventualities is too large to pre-plan for. The solution to deal with these types of problems is an intelligent system that can generate and check plans quickly and under the constraints of the failures.

This intelligent system was designed as a set of independent modules combined with a graphical Java interface to offer an easy way to enter and return information. The modules are written in A-Prolog and model different components of the RCS, as well as the goals that can be achieved by them. The graphical interface is used to input information about the tasks that have been performed by the RCS so far, which faults have been detected, and what tasks should be performed. The tasks that can be performed are to check if the tasks the RCS has already performed have led to a specific goal, or to generate a plan, meaning a set of tasks that satisfies a specific goal in a limited number of steps.

The graphical interface checks if the input is complete, and if so, assembles the appropriate modules into an A-Prolog program Π which it passes to the reasoning system that computes the stable model. The reasoning system that is used in the article are *lparse* and *Smodels*. This approach reduces the checking of a plan to only checking if a model exists for Π with the given plan as input, meaning as a set of facts when looking at it from an A-Prolog perspective.

A planning module is used to generate a set of plans that fulfil a desired goal under the constraints given by the failures of the system. These plans coincide with the stable models of the program. As such, planning is reduced to answer set solving. The graphical interface translates these stable models into a user-friendly, readable set of tasks, and presents it as a set of solutions to the given information. A look at one of the modules should give an idea of how these programs are constructed.

Example 33: The plumbing module PM consists of approximately 40 rules, and models the plumbing system of the RCS. The purpose of the plumbing system is to deliver fuel and oxidisers from the tanks of the space shuttle to its jets. The system consists of tanks, jets, and pipe junctions as well as valves that control the flow from one element to another. PM presents these system as a directed graph in which the tanks, jets, and pipe junctions function as nodes, and the valves function as the connecting edges.

The purpose of PM is to describe the paths that the fuel and oxidisers can take throughout the system, and how those paths are affected by leaks and the status of the valves.

Every jet that is required for a maneuver must have a leak free and open path leading to it for the maneuver to be performed.

The input for the rules that make up *PM* is the structure of the plumbing system, its current faulty components, and a list of all its open valves. *PM* then computes the distribution of pressure throughout the system, which jets are supplied with fuel, and, in turn, which maneuvers can be performed by the space shuttle at the moment.

An example for how this is achieved is through a rule that contains the predicate `pressurizedBy(N1, Tk)`. The rule describes the pressure of a node *N1* that is obtained through a tank *Tk*. There are special nodes that are always pressurized, namely the helium tanks, and for all other nodes this is decided recursively. This means that if a node *N1*, that is not leaking, is pressurized by a tank *Tk* and is connected through an open valve to another node *N2*, *N2* is also pressurized. The way that this rule is written in A-Prolog is:

```
h(pressurizedBy(N1, Tk), T) :-
    not tankOf(N1, R),
    not h(leaking(N1), T),
    link(N2, N1, V),
    h(inState(V, open), T),
    h(pressurizedBy(N2, Tk), T).
```

Code Example 4.5: Pressurized node

The other modules that are part of the system are the valve control module, the extended valve control module, which extends the valve module to also consider electrical circuits, the circuit theory module, which describes the electrical circuits required by the RCS, and the previously mentioned planning module. All modules except for the planning module follow the idea that has been described in the example above, in which they model the current status of the components of their system.

The planning module uses the other modules to check if the current status of the subsystems fulfils a given goal, or, alternatively, which steps need to be taken to fulfil that goal. The goals can often be split into subgoals that need to be fulfilled by each individual system, which, in turn, can be checked individually by the planning module or be assembled into set of steps that must be taken for each subsystem.

Modularity is an integral aspect of the decision support system, which leads to a significant increase in the efficiency of the planning process. Each module can be optimized on its own and can be omitted if it is not required for a specific goal.

The decision support system illustrates the utility of both answer set programming and planning in general. The utilization of answer set programming and A-Prolog in a large knowledge-intensive application was a first for its time, and highlights their efficiency and applicability. This system still serves as one of the most exemplary applications of answer set programming to this day.

CHAPTER 5

Conclusion

Answer set programming is an approach to problem solving that is still evolving. Its many implementations and extensions have greatly increased the scope of problems that it can solve while also increasing its efficiency. Even though there are very few practical applications of answer set programming outside of academical purposes, the ones that do exist offer a good example of what it can achieve.

5.1 Summary

Answer set programming is an effective declarative approach to problem solving. It is deeply rooted in first-order logic and logic programming, and incorporates different methods and concepts derived from both. The core concept of answer set programming is the stable model semantics. The stable model semantics is used to allow for the computation of a solution to a problem.

The canonical language for answer set programming is called A-Prolog or AnsProlog. AnsProlog depends upon grounding, an integral step which contextualizes the program and is performed by applications such as *lpase* and *GrinGo*.

There are many implementations of answer set programming in the form of answer set solvers. One of the first answer set solvers ever created was *Smodels*, which serves as a standard for answer set programming. Another answer set solver which is still used to this day is *Clasp*, which utilizes a different, conflict-driven approach to solving.

One of the most well-known applications of answer set programming is a decision-support system for a space shuttle.

Answer set programming has proven to be a flexible and practical branch of logic programming. Its easy-to-use modelling language as well as the availability of a wide selection of different answer set solvers and grounders makes it highly accessible. This accessibility and effectiveness makes answer set programming a very attractive choice for solving complex problems. Its constant evolution and increasing proposed applications makes it very likely that answer set programming will be utilized even more in the future.

Bibliography

- [1] Mario Alviano, Francesco Calimeri, Carmine Dodari, Davide Fusca, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV 2. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017*, pages 215–221. Springer, Cham, 2017.
- [2] Mario Alviano and Wolfgang Faber. Aggregates in answer set programming. *KI - Künstliche Intelligenz*, 32:119–124, 2018-08.
- [3] Mario Alviano and Nicola Leone. On the properties of GZ-aggregates in answer set programming. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI*, 2016.
- [4] Giovanni Amendola. Towards quantified answer set programming. In Marco Maratea and Mauro Vallati, editors, *Proceedings of the Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion co-located with the Federated Logic Conference, Oxford, United Kingdom, July 13, 2018*, volume 2271 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.
- [5] Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. *Proceedings of ICLP’09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP’09)*, pages 16–30, 2009.
- [6] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules. *International Symposium on Logical Formalization of Commonsense Reasoning*, page 11, 2003.
- [7] M. Barry and R. Watson. Reasoning about actions for spacecraft redundancy management. In *1999 IEEE Aerospace Conference. Proceedings (Cat. No.99TH8403)*, volume 5, pages 101–112. IEEE, 1999.
- [8] Jon Barwise. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier, 1977.
- [9] Marjon Blondeel, Steven Schockaert, Dirk Vermeir, and Martine De Cock. Fuzzy answer set programming: An introduction. In Ronald R. Yager, Ali M. Abbasov, Marek Z. Reformat, and Shahnaz N. Shahbazova, editors, *Soft Computing: State of the Art Theory and Novel Applications*, volume 291, pages 209–222. Springer Berlin Heidelberg, 2013. Series Title: Studies in Fuzziness and Soft Computing.
- [10] Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In Eduardo Fermé and João Leite, editors, *Logics*

- in *Artificial Intelligence*, volume 8761, pages 166–180. Springer International Publishing, 2014. Series Title: Lecture Notes in Computer Science.
- [11] Piero A. Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. Answer set programming. In *A 25-year Perspective on Logic Programming*, volume Volume 6125 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2010.
 - [12] Paul Borchert, Christian Anger, Torsten Schaub, and Mirosław Truszczyński. Towards systematic benchmarking in answer set programming: The dagstuhl initiative. In *Logic Programming and Nonmonotonic Reasoning*, pages 3–7. Springer Berlin Heidelberg, 2004.
 - [13] Annalisa Bossi and Maria Chiara Meo. Theoretical foundations and semantics of logic programming. In Agostino Dovier and Enrico Pontelli, editors, *A 25-Year Perspective on Logic Programming*, volume 6125, pages 15–36. Springer Berlin Heidelberg, 2010. Series Title: Lecture Notes in Computer Science.
 - [14] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54:92–103, 2011-12.
 - [15] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of rule-based query answering. In Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola Henze, Paula-Lavinia Patranjan, and Robert Tolksdorf, editors, *Reasoning Web*, volume 4636 of *Lecture Notes in Computer Science*, pages 1–153. Springer Berlin Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.
 - [16] Duygu Cakmak, Esra Erdem, and Halit Erdogan. Computing weighted solutions in ASP: representation-based method vs. search-based method. *Annals of Mathematics and Artificial Intelligence*, 62:219–258, 2011-07.
 - [17] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artificial Intelligence*, 231:151–181, 2016-02.
 - [18] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *Theory and Practice of Logic Programming*, 14:117–135, 2014-01.
 - [19] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, (eds Gallaire and Minker), 1978., pages 293–322. Plenum Press, 1978.
 - [20] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *Inductive Logic Programming*, volume 7207, pages 91–97. Springer Berlin Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.
 - [21] F M Donini, M Lenzerini, D Nardi, F Pirri, and M Schaerf. Nonmonotonic reasoning. *Artificial Intelligence Review*, 4:163–210, 1990.
 - [22] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689, pages 40–110. Springer Berlin Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

- [23] Wolfgang Faber. An introduction to answer set programming and some of its extensions. In *Manna M., Pieris A. (eds) Reasoning Web. Declarative Artificial Intelligence. Reasoning Web 2020.*, volume 12258 of *Lecture Notes in Computer Science*. Springer, Cham, 2020.
- [24] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *LPNMR 2011: Logic Programming and Nonmonotonic Reasoning*, pages 54–66. Springer Berlin Heidelberg, 2011.
- [25] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
- [26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo, 2018-03-20.
- [27] Martin Gebser, Benjamin Kaufmann, Andre Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, page 386, 2007.
- [28] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 4483, pages 260–265. Springer Berlin Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.
- [29] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012-08.
- [30] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the seventh answer set programming competition. In Marcello Balduccini and Tomi Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 10377, pages 3–9. Springer International Publishing, 2017. Series Title: Lecture Notes in Computer Science.
- [31] Martin Gebser, Marco Maratea, and Francesco Ricca. The seventh answer set programming competition: Design and results. *Theory and Practice of Logic Programming*, 20:176–204, 2020-03.
- [32] Martin Gebser, Torsten Schaub, and Sven Thiele. *GrinGo: A New Grounder for Answer Set Programming*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer Berlin Heidelberg, 2007.
- [33] Michael Gelfond. Representing knowledge in a-prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408, pages 413–451. Springer Berlin Heidelberg, 2002. Series Title: Lecture Notes in Computer Science.
- [34] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
- [35] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991-08.
- [36] Michael Gelfond and Richard Watson. On methodology of representing knowledge in dynamic domains. *Science of Computer Programming*, 42:87–99, 2002-01.

- [37] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155:1549–1561, 2007-06.
- [38] Steffen Hölldobler and Lukas Schweizer. Answer set programming and CLASP – a tutorial. In *Young Scientists’ International Workshop on Trends in Information Processing*, volume 1145 of *YSIP*, pages 77–95, 2014.
- [39] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37:25–32, 2016-10-07.
- [40] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31:38–43, 1988-01.
- [41] Ryan Lawrence. Efficient algorithms for clause-learning SAT solvers. Master’s thesis, Simon Fraser University, Canada, 2004.
- [42] Yuliya Lierler. cmodels – SAT-based disjunctive answer set solver. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 3662, pages 447–451. Springer Berlin Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.
- [43] Vladimir Lifschitz. Action languages, answer sets, and planning. In Krzysztof R. Apt, Victor W. Marek, Mirek Truszczynski, and David S. Warren, editors, *The Logic Programming Paradigm*, pages 357–373. Springer Berlin Heidelberg, 1999. Series Title: Artificial Intelligence.
- [44] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 3, pages 1594–1597. AAAI Press, 2008.
- [45] Chris Martens. Notes on answer set programming. CSC 791 Generative Methods for Game Design, 2017.
- [46] Matthew W Moskwicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.
- [47] Ilkka Niemela, Patrik Simons, and Tommi Syrjanen. Smodels: A system for answer set programming. *ArXiv*, cs.AI/0003033, 2000-03-08.
- [48] Ilkka Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53:313–329, 2008-08.
- [49] Ilkka Niemelä and Patrik Simons. Smodels — an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265, pages 420–429. Springer Berlin Heidelberg, 1997. Series Title: Lecture Notes in Computer Science.
- [50] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Lecture Notes in Computer Science*, vol 1730, pages 317–331. Springer, Berlin, Heidelberg, 1999.
- [51] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990, pages 169–183. Springer Berlin Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

- [52] Teodor C. Przymusiński. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991-08.
- [53] R. Reiter. Nonmonotonic reasoning. *Annual Review of Computer Science*, 2:147–186, 1987.
- [54] Raymond Reiter. *On Closed World Data Bases*, pages 55–76. Springer US, Boston, MA, 1978.
- [55] Torsten Schaub. Answer set programming in a nutshell. <https://www.cs.uni-potsdam.de/~torsten/Potassco/Talks/nutshellCSPSAT15.pdf>, 2021. Lecture notes, Beyond Satisfiability, University of Potsdam.
- [56] Marek Sergot. Stratified logic programs. https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Stratified_491-2x1.pdf, 2005. Lecture notes, Knowledge Representation, Department of Computing Imperial College, London.
- [57] Marek Sergot. Negation as failure (normal logic programs). https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/NBF_491-2x1.pdf, 2010. Lecture notes, Knowledge Representation, Department of Computing Imperial College, London.
- [58] Patrik Simons. Extending the stable model semantics with more expressive rules. In *Lecture Notes in Computer Science*, vol 1730, pages 305–316. Springer, Berlin, Heidelberg, 1999.
- [59] Patrik Simons and Ilkka Niemelä. Evaluating an algorithm for default reasoning. In *Working Notes of the IJCAI’95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems*, pages 66–72, 1995-08.
- [60] Patrik Simons, Ilkka Niemelä, and Timo Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002-06.
- [61] Timo Soinen and Ilkka Niemela. Formalizing configuration knowledge using rules with choices. *Seventh International Workshop on Nonmonotonic Reasoning, Trento*, page 11, 1998.
- [62] Tommi Syrjänen. Lparse 1.0 user’s manual. <http://www.tcs.hut.fi/Software/smodels/>, 2001.
- [63] Mirek Truszczyński and Victor W. Marek. Stable models and an alternative logic programming paradigm. In *The Logic programming paradigm: a 25-year perspective*, pages 169–181. Springer-Verlag, 1999.
- [64] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976-10.
- [65] Richard Watson. An application of action theory to the space shuttle. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551, pages 290–304. Springer Berlin Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.
- [66] Hantao Zhang and Mark Stickel. Implementing the davis putnam method. *Journal of Automated Reasoning*, 24:277–296, 2000.