5-2010

SAT-based Answer Set Programming

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

The Dissertation Committee for Yuliya Lierler
certifies that this is the approved version of the following dissertation:

SAT-based Answer Set Programming

Committee:

Vladimir Lifschitz, Supervisor

Robert Boyer

Anna Gal

Peter Stone

Mirosław Truszczyński

SAT-based Answer Set Programming

by

Yuliya Lierler, B.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2010

To my grandparents Nina and Nikolay for their love

Acknowledgments

I have had the privilege to be advised by Vladimir Lifschitz. I would like to thank him for his continuous advice, support, insight, and encouragement throughout my work on this dissertation. During the many years that I have known and worked with Vladimir he has been an advisor, a teacher, a colleague, and a friend to me. I am thankful to him for serving these many roles. For years Vladimir helped me to discover the world of science and guided me in this world. I am grateful for the thousands of discussions in person and via email that we have had. Vladimir's ability to think deeply, notice details, and express himself clearly, both in speaking and writing, fascinates me till this moment. I am also thankful to his wife Elena Lifschitz for her care. My decision to apply for a PhD program came by chance from a conversation that I once had with Vladimir and Elena Lifschitz. I am grateful to both of them for opening the door to science for me.

I am thankful to the other members of my dissertation committee: Robert Boyer, Anna Gal, Peter Stone, and Mirosław Truszczyński for serving on my committee and for their useful comments on my dissertation.

It was a privilege to be in a research group with many wonderful graduate students and I extend my thanks to Esra Erdem, Selim Erdoğan, Paolo Ferraris, Joohyung Lee, Wanwan Ren and Fangkai Yang. I want to especially thank Selim Erdoğan for all the encouragements and support he has been giving me through the years of our friendship.

The University of Texas at Austin has almost become my home. It has been a great place to study and work and I will miss it a lot. I am thankful to all my teachers, colleagues, and friends who have made these years fruitful, challenging, interesting, and fun.

My academic journey started at the Belarusian State University of Informatics and Radioelectronics where I had the chance to study with a group (635701) of great students many of whom are still my friends. I wish to thank the head of the AI department Golenkov V.V. and the AI Professor Sharaya N.A. for setting exemplary educational program and their dedication to science.

I am thankful to Günther Görz for advising me and introducing me to the exciting world of natural language understanding and computational semantics.

For years I have benefited from many delightful scientific and personal discussions with my colleagues by science Marcello Balduccini, Chitta Baral, Johan Bos, Martin Brain, Gerhard Brewka, Pedro Cabalar, Stefania Costantini, Thomas Eiter, Wolfgang Faber, Martin Gebser, Gregory Gelfond, Michael Gelfond, Enrico Giunchiglia, Tomi Janhunen, Nicola Leone, Bernd Ludwig, Marco Maratea, Bernhard Nebel, Ilkka Niemelä, Gerald Pfeifer, Peter Reiss, Torsten Schaub, Bernhard Schiemann, Iman Thabet, Son Tran, Mirek Truszczyński, Marina De Vos, Stefan Woltran. I am grateful to all of them.

I am in debt to my friends Gurucharan Huchachar, Aram Karakhanyan, Elina Drayevskaya, John Beavers, Mikhail Bilenko, Ali Amjad Khoja, Yuliya Volkovinskaya, Boris Olesiuk, Olga Nobst, and Inna Imayeva for being important part of my life. Guru, in spite of thousands miles between us, has been part of my daily life through thousands of emails.

The greatest support for everything I have done in my life come from my parents, Natalya Bogataya and Konstantin Bobovich. I thank them for all their continuous love, care, and encouragement. They have been by my side every single

step that I have made and yet they allowed me complete freedom to define my path. I also wish to thank my grandparents, Nina and Nikolay Bogaty. They still see me and treat me as a child. My brother and sister-in-law, Sergey and Lena Babovich, amaze me by their ability to enjoy life, be positive, happy, and engage themselves into numerous new activities. Hildegard and Heinz Lierler took me to their heart. My aunt Nadya has been my dearest friend who kept my spirits high, even with the many miles between us. My cousin Sasha has been my little companion whenever I had a chance to be in Belarus. My uncle Kolia helped me to settle in two houses that I have lived in on the way to this dissertation.

I thank my husband Frank who has moved thousands of miles from his home country to allow me to proceed on my path towards this dissertation. He also spent hours proofreading its text. I am grateful for all his love, care, and support. My deepest love goes to our first baby Nina who brought new light of discovery and joy in my life.

Yuliya Lierler

The University of Texas at Austin
May 2010

SAT-based Answer Set Programming

Publication No. _____

Yuliya Lierler, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Vladimir Lifschitz

Answer set programming (ASP) is a declarative programming paradigm oriented towards difficult combinatorial search problems. Syntactically, ASP programs look like Prolog programs, but solutions are represented in ASP by sets of atoms, and not by substitutions, as in Prolog. Answer set systems, such as SMODELS, SMODELS$_{cc}$, and DLV, compute answer sets of a given program in the sense of the answer set (stable model) semantics. This is different from the functionality of Prolog systems, which determine when a given query is true relative to a given logic program. ASP has been applied to many areas of science and technology, from the design of a decision support system for the Space Shuttle to graph-theoretic problems arising in zoology and linguistics.

The "native" answer set systems mentioned above are based on specialized

search procedures. Usually these procedures are described fairly informally with the use of pseudocode. We propose an alternative approach to describing algorithms of answer set solvers. In this approach we specify what "states of computation" are, and which transitions between states are allowed. In this way, we define a directed graph such that every execution of a procedure corresponds to a path in this graph. This allows us to model algorithms of answer set solvers by a mathematically simple and elegant object, graph, rather than a collection of pseudocode statements. We use this abstract framework to describe and prove the correctness of the answer set solver SMODELS, and also of SMODELS$_{cc}$, which enhances the former with learning and backjumping techniques.

Answer sets of a tight program can be found by running a SAT solver on the program's completion, because for such a program answer sets are in a one-to-one correspondence with models of completion. SAT is one of the most widely studied problems in computational logic, and many efficient SAT procedures were developed over the last decade. Using SAT solvers for computing answer sets allows us to take advantage of the advances in the SAT area. For a nontight program it is still the case that each answer set corresponds to a model of the program's completion but not vice versa. We show how to modify the search method typically used in SAT solvers to allow testing models of completion and employ learning to utilize testing information to guide the search. We develop a new SAT-based answer set solver, called CMODELS, based on this idea.

We develop an abstract graph based framework for describing SAT-based answer set solvers and use it to represent the CMODELS algorithm and to demonstrate its correctness. Such representations allow us to better understand similarities and differences between native and SAT-based answer set solvers. We formally compare the SMODELS algorithm with a variant of the CMODELS algorithm without learning.

Abstract frameworks for describing native and SAT-based answer set solvers

facilitate the development of new systems. We propose and implement the answer set solver called SUP that can be seen as a combination of computational ideas behind CMODELS and SMODELS. Like CMODELS, the solver SUP operates by computing a sequence of models of completion for the given program, but it does not form the completion. Instead, SUP runs the *Atleast* algorithm, one of the main building blocks of the SMODELS procedure. Both systems CMODELS and SUP, developed in this dissertation, proved to be a competitive answer set programming systems.

Contents

Chapter 1

Introduction

Answer set programming (ASP) is a declarative programming paradigm oriented towards difficult combinatorial search problems [Marek and Truszczyński, 1999; Niemelä, 1999]. Syntactically, ASP programs look like Prolog programs, but solutions are represented in ASP by sets of atoms, and not by substitutions, as in Prolog. Answer set systems, such as SMODELS[1], SMODELS$_{cc}$[2] [Ward and Schlipf, 2004] and DLV[3], compute answer sets of a given program in the sense of the answer set (stable model) semantics [Gelfond and Lifschitz, 1988; 1991]. ASP has been applied to many areas of science and technology.

The "native" answer set systems mentioned above are based on specialized search procedures. Usually computation procedures are described in terms of pseudocode. In [Nieuwenhuis *et al.*, 2006], the authors proposed an alternative approach to describing DPLL-like procedures commonly used in SAT solvers [Gomes *et al.*, 2008]. They introduced an abstract framework that captures what "states of computation" are, and what transitions between states are allowed. In this way, [Nieuwenhuis *et al.*, 2006] defines a directed graph such that every execution of the DPLL procedure corresponds to a path in this graph. Some edges may correspond to unit propagation steps, some to branching, some to backtracking. This allows the authors to model a DPLL-like algorithm by a mathematically simple and elegant object, graph, rather than a collection of pseudocode statements. Such an abstract way of presenting algorithms simplifies the analysis of their correctness and facilitates for-

[1] http://www.tcs.hut.fi/Software/smodels/ .
[2] http://www.nku.edu/%7Ewardj1/Research/smodels_cc.html .
[3] http://www.dbai.tuwien.ac.at/proj/dlv/ .

mal reasoning about their properties. In this dissertation, we extend this framework for describing algorithms of answer set solvers. We use this abstract framework to represent and prove the correctness of the answer set solver SMODELS, and also of SMODELS$_{cc}$, which enhances the former with learning and backjumping techniques.

The answer set systems mentioned above are based on specialized search procedures. For the large class of *tight* programs [Fages, 1994] we can use SAT solvers for finding their answer sets, because the answer sets of a tight program coincide with models of its *completion* in the sense of [Clark, 1978]. SAT is one of the most widely studied problems in computational logic, and many efficient SAT procedures were developed over the last decade employing such sophisticated techniques as backjumping and learning. Using SAT solvers for computing answer sets allows us to take advantage of the advances in the SAT area. The main topic of this dissertation is developing a SAT-based approach for finding answer sets of a program. We designed a new answer set solver CMODELS[4] that is based on the following ideas: First, it converts the given logic program into a propositional formula using completion. Second, CMODELS clausifies the completion. Third, a satisfiability solver is applied to the resulting formula.

For a nontight program it is still the case that each answer set corresponds to a model of the program's completion but not necessarily the other way around. Lin and Zhao [2002] introduced *loop formulas* so that any model of completion that satisfies all loop formulas of a program is also an answer set. The straightforward approach of using SAT solvers on the program's completion extended by its loop formulas for finding answer sets of a program is unfortunately not feasible, because the number of loop formulas may be exponential. Nevertheless, the system ASSAT[5] proposed an algorithm that enumerates loop formulas "on demand" for finding answer sets by means of SAT solvers. We propose another algorithm that also exploits loop formulas but in a more sophisticated way by utilizing learning techniques available in most modern SAT solvers and use it to extend the answer set solver CMODELS to nontight programs. CMODELS operates by computing a model of completion of the given logic program, and then continuing this process, if necessary, in the presence of additional constraints expressed by loop formulas.

The answer set semantics was extended to programs with more general syn-

[4] http://www.cs.utexas.edu/users/tag/cmodels .
[5] http://assat.cs.ust.hk/ .

tax than allowed by Prolog. For instance, choice and weight constraint rules were introduced in [Niemelä and Simons, 2000]. These rules often allow more concise encoding but the problem of deciding whether a program with such rules has an answer set is still NP-complete. In [Gelfond and Lifschitz, 1991] programs with disjunctive rules were introduced. The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete [Eiter and Gottlob, 1993].

In this dissertation we demonstrate the possibility of using SAT solvers for finding answer sets of programs with extended syntax. Until recently there were only two answer set systems that allowed programs with disjunctive rules DLV and GNT[6] [Janhunen *et al.*, 2006]. We extend the SAT-based approach to disjunctive programs that also allow weight and choice constraints and implement this extension in CMODELS. To the best of our knowledge CMODELS is the only answer set solver that allows combination of these rules, although DLV accepts programs with similar constructs. CMODELS is a relatively new system, but it has been already used in various application domains such as wire-routing [Erdem and Wong, 2004], reconstruction of phylogenies [Brooks *et al.*, 2007], formal verification of abstract state machines [Tang and Ternovska, 2005] and planning [Son *et al.*, 2005]. In the area of phylogenies reconstruction CMODELS outperformed specialized tools and helped to discover previously unknown results.

We develop an abstract graph based framework for describing SAT-based answer set solvers. We use this framework to represent the CMODELS algorithm and to demonstrate its correctness. Such representation allows us to better understand the similarities and differences between native and SAT-based answer set solvers. We formally compare the SMODELS algorithm with a variant of the CMODELS algorithm without learning.

Design of abstract frameworks for describing native and SAT-based answer set solvers allows a clear high-level view on algorithms. This facilitates the development of new systems. We propose and implement the answer set solver called SUP[7] that can be seen as a combination of computational ideas behind CMODELS and SMODELS. The solver SUP operates in a similar way to CMODELS, by computing a sequence of models of completion of the given program, but it does not form the completion. Instead, SUP runs the *Atleast* algorithm, one of the main building

[6] http://www.tcs.hut.fi/Software/gnt/ .
[7] http://www.cs.utexas.edu/users/tag/sup .

blocks of the SMODELS procedure. Both systems CMODELS and SUP developed in the course of this dissertation proved to be competitive answer set solvers.

The dissertation is organized as follows. Chapter 2 introduces the answer set programming paradigm, and spans through the area's history, its systems, and applications. Chapter 3 provides a review of propositional satisfiability. It also presents an abstract framework introduced in [Nieuwenhuis *et al.*, 2006] for describing the basic satisfiability algorithm DPLL and its extensions.

Chapter 4 defines traditional logic programs and introduces the notions of completion and tightness that form the foundation of this work. Chapter 5 defines an abstract framework for describing answer set solvers similar to the one introduced in Chapter 3 for the DPLL procedure. We also represent the algorithm of the answer set solver SMODELS by means of this framework and demonstrate its correctness. Chapter 6 introduces the SAT-based method for finding answer sets for tight traditional programs.

In Chapters 7 and 8 we discuss the extension of our method to more general programs. Chapter 7 introduces choice and weight rules. Chapter 8 explains the methodology of applying SAT solvers for finding answer sets of the programs that use these syntactic features.

Chapter 9 introduces the concept of a loop formula [Lin and Zhao, 2002]. In Chapter 10 we define an abstract framework for a "generate and test" DPLL algorithm similar to the one introduced in Chapter 3 for the DPLL procedure. Using this abstract framework and the loop formula concept, in Chapter 11 we develop an algorithm that applies SAT solvers for finding answer sets for nontight programs. We prove the correctness of the new SAT-based answer set solver algorithm.

Chapter 12 defines an abstract framework for describing native answer set solver algorithms with learning. In particular, it provides the description of the algorithms behind systems SMODELS$_{cc}$ and SUP. The development of this abstract framework promoted the design of the system SUP. At the end of the chapter we demonstrate experimental results for this system.

Chapter 13 starts by defining disjunctive logic programs and the notions of completion, tightness, and loop formulas for these programs. Then, it extends the SAT-based method for finding answer sets introduced in Chapter 9 to disjunctive programs.

In Chapter 14 we discuss the work by other researchers related to the ap-

proach presented in the dissertation. We conclude with Chapter 15 that summarizes the contents of this dissertation.

Chapter 2

Answer Set Programming

As discussed in the introduction, answer set programming (ASP) is a new programming paradigm introduced in [Marek and Truszczyński, 1999; Niemelä, 1999]. It is a form of declarative programming related to logic programming languages, such as Prolog. The input of Prolog consists of a logic program and a query. In ASP, solutions to a problem are represented by answer sets, and not by answer substitutions produced in response to a query as in Prolog. Instead of Prolog systems, this programming method first uses grounders, such as LPARSE[1] [Syrjanen, 2003] and GRINGO[2] [Gebser *et al.*, 2007a], to instantiate the variables in a given program, and then applies answer set solvers, such as SMODELS [Simons and Syrjaenen, 2007] and DLV [Eiter *et al.*, 1998], to generate answer sets. The systems interpret logic programs via the answer set semantics [Gelfond and Lifschitz, 1988; 1991; Niemelä and Simons, 2000]. This approach is similar to propositional satisfiability checking, where a propositional formula encodes the given problem and the models of the formula correspond to solutions. The model generation approach in place of query evaluation is the most characteristic feature of answer set programming. ASP is oriented towards search problems occurring in the area of knowledge representation and reasoning.

[1] http://www.tcs.hut.fi/Software/smodels/ .
[2] http://sourceforge.net/projects/gringo/ .

2.1 A Sample Program

Here we illustrate the answer set programming paradigm at work. We take the problem of coloring a graph for this purpose: *Consider a graph given as a set of nodes and edges; Find a way to color the nodes with n colors such that no two adjacent nodes are colored with the same color.*

The system LPARSE [Syrjanen, 2003] is a front-end for a number of answer set solvers including SMODELS and CMODELS. It takes a logic program containing variables in its rules, and outputs the grounded version of the program, i.e., the program with variables instantiated by constants. Here is the program *color.lp* that encodes the graph coloring problem:

```
% color.lp by Ilkka Niemelae
%
% Facts nodeColor(V,C) in an answer set provide
% a coloring of the graph.

nodeColor(N,C):-  node(N), clr(C), not diffColor(N,C).
diffColor(N,C) :- node(N), clr(C), clr(C1), C != C1, nodeColor(N,C1).

:- edge(N1,N2), clr(C), nodeColor(N1,C), nodeColor(N2,C).

% colors
clr(1). clr(2). clr(3).

% graph description
node(a). node(b).
node(c). node(d).
edge(a,b). edge(b,c).
edge(c,d). edge(d,a).
```

Just as a set of clauses can have many models, a logic program can have many answer sets. The main idea of answer set programming is that these answer sets encode solutions to the problem. For instance, for the program *color.lp* every answer set corresponds to a solution. Consider the predicate $nodeColor(N, C)$. It

expresses that node N has color C. Given an answer set of *color.lp* program, the atoms from this set that contain the predicate $nodeColor(N, C)$ provide information on how each node of the graph is colored.

Let us now discuss the intuition behind the rules in program *color.lp*. Consider the rule

```
nodeColor(N,C):-  node(N), clr(C), not diffColor(N,C).
```

It states that a node N is assigned a color C unless C differs from the color assigned to N. The rule

```
diffColor(N,C) :- node(N), clr(C), clr(C1), C != C1, nodeColor(N,C1).
```

says that color C differs from a color assigned to a node N. These rules guarantee that only one color is assigned to every node. We may describe these rules as the rules that generate "candidate models". On the other hand, the rule

```
:- edge(N1,N2), clr(C), nodeColor(N1,C), nodeColor(N2,C).
```

tests candidate models on the condition that no adjacent nodes share the same color. The syntax and semantics of programs like this will be discussed in Chapter 4.

The system SMODELS computes an answer set for this program as follows:

```
% lparse color.lp | smodels
smodels version 2.34. Reading...done
Answer: 1
Stable Model:
nodeColor(d,3) nodeColor(b,3) nodeColor(c,2) nodeColor(a,1)
edge(d,a) edge(c,d) edge(b,c) edge(a,b)
node(d) node(c) node(b) node(a) clr(3) clr(2) clr(1)
diffColor(c,3) diffColor(a,3) diffColor(d,2) diffColor(b,2)
diffColor(a,2) diffColor(d,1) diffColor(c,1) diffColor(b,1)
Duration: 0.004
```

2.2 Answer Set Programming Applications

In general, answer set programming can be seen as a generic combinatorial reasoning and search paradigm. It is based on an implementation-independent declarative

semantics. This makes it easier to develop various applications, as the internal implementation aspects are hidden within an answer set solver. Although answer set programming is a new programming approach, efficient answer set solvers such as SMODELS [Niemelä and Simons, 1996; Simons *et al.*, 2002; Simons and Syrjaenen, 2007], DLV [Eiter *et al.*, 1997; Leone and et al., 2005], and CMODELS (this dissertation) allowed the answer set programming paradigm to be successfully applied in various domains including

- planning [Dimopoulos *et al.*, 1997; Lifschitz, 1999; Son *et al.*, 2005],
- space shuttle control [Nogueira *et al.*, 2001],
- reachability analysis [Heljanko, 1999],
- bounded model checking [Liu *et al.*, 1998; Heljanko and Niemelä, 2003],
- logical cryptanalysis [Hietalahti *et al.*, 2000],
- network inhibition [Aura *et al.*, 2000],
- reasoning about policies [Son and Lobo, 2001],
- combinatorial auctions [Baral and Uyan, 2001],
- diagnosis [Eiter *et al.*, 1999; Gelfond and Galloway, 2001; Balduccini and Gelfond, 2003],
- wire-routing [Erdem and Wong, 2004],
- protocol (in)security [Armando *et al.*, 2004],
- query answering [Baral *et al.*, 2005; Tari and Baral, 2005; Nouioua and Nicolas, 2006],
- reconstruction of phylogenies [Brooks *et al.*, 2007],
- formal verification of abstract state machines [Tang and Ternovska, 2005],
- machine code optimization [Brain *et al.*, 2006].

Chapter 3

Propositional Satisfiability Solvers

This chapter is an introduction to the field of propositional satisfiability that plays an important role in this work where we propose to use a satisfiability solver as a search engine for an answer set programming system. For instance, the answer set system CMODELS that implements our approach can use various state-of-the-art SAT solvers for search.

Propositional satisfiability (SAT) is one of the most intensely studied fields in computational logic. Satisfiability is the problem of determining if the variables of a given propositional formula can be assigned truth values in such a way that the formula is evaluated to *True*.

During the last decade, efficient SAT solvers, such as SATZ [Li and Anbulagan, 1997], CHAFF [Moskewicz *et al.*, 2001]), and MINISAT [Een and Biere, 2005] were created. Although all known algorithms have exponential run time in the worst case, SAT solvers have many important applications. Modern SAT solvers often solve hard structured problem instances that involve several million of constraints and over a million of variables [Gomes *et al.*, 2008].

In Section 3.1 we review a pseudo code of the Davis-Putnam-Logemann-Loveland (DPLL) procedure. Section 3.2 presents DPLL by means of an abstract framework introduced in [Nieuwenhuis *et al.*, 2006]. Later in this dissertation we will first modify this abstract DPLL framework to describe SMODELS [Simons, 2000], one of the best known algorithms for finding answer sets of a program; and sec-

ond generalize abstract DPLL framework to underline SAT-based answer set solving method argued for in this work. Section 3.3 provides a brief introduction of such advanced techniques commonly used in SAT solvers as restarts and conflict driven backjumping and learning. In Section 3.4 we will extend the abstract framework presented in Section 3.2 to capture the ideas behind backjumping and learning.

3.1 DPLL

Most modern SAT solvers are based on variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis *et al.*, 1962]. DPLL employs a systematic backtracking search procedure to explore the variable assignments looking for a satisfying assignment. Let us recall that a *literal* is a propositional atom possibly preceded by the classical negation symbol $\neg$, a *clause* is a disjunction of literals, a *unit* clause is a clause that consists of a single literal. By $\overline{l}$ we denote a literal complementary to literal l. Our notation below follows [Gomes *et al.*, 2008]. We identify a formula in conjunctive normal form with the set of clauses corresponding to its conjunctive members. A *partial assignment* is a consistent set of propositional literals. We identify a partial assignment ρ with the conjunction of its elements, and also with the function that maps to *True* the atoms that occur in ρ positively, and to *False* the atoms that occur in ρ negatively. A literal l is *unassigned by* a partial assignment if neither l nor its complement $\overline{l}$ belongs to it. A formula F is in *conjunctive normal form* (CNF) if it is a conjunction of clauses.

For a partial assignment ρ and a CNF formula F, $F|_\rho$ denotes the formula obtained from F by replacing the atoms occurring in ρ with their specified values, and then simplifying the result by removing each clause containing at least one true literal, and deleting all false literals from the remaining clauses. It is clear that for any atom A and any formula $F(A)$, $F(A)|_A$ is equivalent to $F(\mathit{True})$, and $F(A)|_{\neg A}$ is equivalent to $F(\mathit{False})$.

The algorithms DPLL (Algorithm 1) and UNIT-PROPAGATE (Algorithm 2) are reproduced almost verbatim from [Gomes *et al.*, 2008, Section 2.2.1]. The main difference is that we state the specification at the beginning of each algorithm more precisely.

The algorithm UNIT-PROPAGATE is nondeterministic, because the unit clause x can be chosen, generally, in many different ways. The algorithm DPLL is non-

```
DPLL(F, ρ)
Arguments : set F_0 of clauses and a partial assignment ρ_0 such that no atom
            occurs both in F_0 and ρ_0
Value     : SAT, if F_0 ∧ ρ_0 is satisfiable; UNSAT, otherwise
Output    : a partial assignment ρ such that ρ ⊨ ρ_0 ∧ F_0, if F_0 ∧ ρ_0 is
            satisfiable;
no output, otherwise
begin
    (F, ρ) ← UNIT-PROPAGATE(F, ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    l ← a literal such that its atom occurs in F
    if DPLL(F|_l, ρ ∪ {l}) = SAT then return SAT
    return DPLL(F|_l̄, ρ ∪ {l̄})
end
```

Algorithm 1: DPLL

```
UNIT-PROPAGATE(F,ρ)
Arguments : set F_0 of clauses and a partial assignment ρ_0 such that no atom
            occurs both in F_0 and ρ_0
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|_x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

Algorithm 2: UNIT-PROPAGATE

deterministic as well, because UNIT-PROPAGATE is nondeterministic, and because the literal l can be chosen, generally, in many different ways. The procedure UNIT-PROPAGATE performs simplifications on a set of clauses. Consider, for instance, the invocation of UNIT-PROPAGATE on the set of clauses $\{a, b \vee \neg a\}$ and the empty partial assignment. It will return an empty set of clauses and a partial assignment consisting of $\{a, b\}$.

In order to find a satisfying partial assignment for a CNF formula F, DPLL is initially invoked with the formula F and the empty partial assignment ρ. The idea behind DPLL is to select repeatedly an unassigned literal l in the input formula F and to recursively search for a satisfying partial assignment for $F|_l$ and $F|_{\overline{l}}$. The step where such an l is chosen is commonly referred to as a branching step. Setting l

to *True* or *False* when making a recursive call is called a *decision*. The end of each recursive call that takes F back to fewer assigned variables, is called the backtracking step.

For instance, let F be the set consisting of the clauses

$$a \vee b$$
$$\neg a \vee c.$$

In order to find a satisfying partial assignment for F, DPLL is initially invoked with F and the empty partial assignment. It is then possible that a recursive call to DPLL$(c, \{a\})$ is made. UNIT-PROPAGATE$(c, \{a\})$ returns $(\{\}, \{a, c\})$. Since the set of clauses is empty after the invocation of UNIT-PROPAGATE, DPLL$(c, \{a\})$ outputs $\{a, c\}$ as a satisfying partial assignment and returns SAT. Consequently, DPLL$(F, \emptyset)$ also returns SAT.

3.2 Abstract DPLL

As mentioned in the previous section, most modern SAT solvers implement enhancements of the DPLL procedure. Usually these enhancements of DPLL are described fairly informally with the use of pseudocode. It is often difficult to understand the precise meaning of these modifications and to prove their properties on the basis of such informal descriptions. In [Nieuwenhuis *et al.*, 2006], the authors proposed an alternative approach to describing DPLL and its enhancements (for instance, backjumping and learning discussed in Sections 3.3 and 3.4). They describe variants of DPLL by means of transition systems that can be viewed as an abstract framework underlying the DPLL computation.

This section of the dissertation presents DPLL by means of an abstract framework. In Section 5.2 we adopt an abstract framework for describing the SMODELS algorithm [Simons, 2000] for finding answer sets of a program. We then will generalize an abstract DPLL framework to underline the SAT-based answer set solving method argued for in this dissertation.

The abstract framework introduced in [Nieuwenhuis *et al.*, 2006] describes what "states of computation" are, and which transitions between states are allowed. In this way, it defines a directed graph such that every execution of the DPLL procedure corresponds to a path in this graph. Some edges may correspond to unit prop-

agation steps, some to branching, some to backtracking. This allows us to model a DPLL algorithm by a mathematically simple and elegant object, graph, rather than a collection of pseudocode statements. Such an abstract way of presenting DPLL simplifies the analysis of its correctness and facilitates formal reasoning about its properties. Instead of reasoning about pseudocode constructs, we can reason about properties of a graph. For instance, by proving that the graph corresponding to a version of DPLL is acyclic we demonstrate that the algorithm always terminates. On the other hand, by checking that every terminal state corresponds to a solution we establish the correctness of the algorithm.

The graph introduced in [Nieuwenhuis *et al.*, 2006] is actually an imperfect representation of DPLL in the sense that some paths in the graph do not correspond to any execution of DPLL (for example, paths in which branching is used even though unit propagation is applicable). But this level of detail is irrelevant when we talk about correctness. Furthermore, it makes the correctness theorems more general. These theorems cover not only executions of the pseudo-code, but also some computations that are prohibited by its details.

We start by reviewing the abstract framework for DPLL developed in [Nieuwenhuis *et al.*, 2006] in a form convenient for our purposes, as in [Lierler, 2008].

For a set σ of atoms, a *record* M relative to σ is a list of literals over σ where

(i) some literals in M are annotated by Δ that marks them as *decision* literals,

(ii) M contains no repetitions.

The concatenation of two such lists is denoted by juxtaposition. Frequently, we consider a record as a set of literals, ignoring both the annotations and the order between its elements. A literal l is *unassigned by* a record if neither l nor its complement $\overline{l}$ belongs to it.

A *state* relative to σ is either a distinguished state *FailState* or a record relative to σ. For instance, the states relative to a singleton set $\{a\}$ of atoms are

$$\begin{array}{l} \mathit{FailState},\ \ \emptyset,\ \ a,\ \neg a,\ \ a^{\Delta},\ \ \neg a^{\Delta}, a\ \neg a,\ \ a^{\Delta}\ \neg a, \\ a\ \neg a^{\Delta},\ \ a^{\Delta}\ \neg a^{\Delta}, \neg a\ a,\ \ \neg a^{\Delta}\ a,\ \ \ \neg a\ a^{\Delta},\ \ \neg a^{\Delta}\ a^{\Delta}, \end{array}$$

where by $\emptyset$ we denote the empty list.

If C is a disjunction (conjunction) of literals then by $\overline{C}$ we understand the

Unit Propagate:

$$M \implies M\ l \text{ if } \begin{cases} C \vee l \in F \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

Decide:

$$M \implies M\ l^{\Delta} \text{ if } \begin{cases} M \text{ is consistent and} \\ l \text{ is unassigned by } M \end{cases}$$

Fail:

$$M \implies \mathit{FailState} \text{ if } \begin{cases} M \text{ is inconsistent and} \\ M \text{ contains no decision literals} \end{cases}$$

Backtrack:

$$P\ l^{\Delta}\ Q \implies P\ \bar{l} \text{ if } \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$$

Figure 3.1: The transition rules of the graph DP_F.

conjunction (disjunction) of the complements of the literals occurring in C. We will sometimes identify a conjunction (disjunction) with the multiset of its elements. Given multisets X and Y, by $X \subseteq Y$ we denote that every element of X is also an element of Y. (This is different from standard definition of $\subseteq$ for multisets[1] which takes into account the multiplicity of elements.)

For any CNF formula F (a finite set of clauses), we will define the *DPLL graph* DP_F. The nodes of DP_F are the states relative to the set of atoms occurring in F. We use the terms "state" and "node" interchangeably. Recall that a node is called *terminal* in a graph if there is no edge leaving this node in the graph. If a state is consistent and complete then it represents a truth assignment for F.

The set of edges of DP_F is described by a set of "transition rules." Each transition rule is an expression $M \implies M'$ followed by a condition, where M and M' are nodes of DP_F. Whenever the condition is satisfied, the graph contains an edge from node M to M'. Figure 3.1 presents four transition rules that characterize the edges of DP_F.

This graph can be used for deciding the satisfiability of a formula F simply by constructing an arbitrary path leading from node $\emptyset$ until a terminal node M is

[1] http://en.wikipedia.org/wiki/Multiset

reached. The following theorem shows that this process always terminates, that F is unsatisfiable if M is *FailState*, and that M is a model of F otherwise.

Theorem 1. *For any CNF formula F,*

(a) graph DP_F is finite and acyclic,

(b) any terminal state of DP_F other than FailState is a model of F,

(c) FailState is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable.

For instance, let F be the set consisting of the clauses

$$a \vee b$$
$$\neg a \vee c.$$

Here is a path in DP_F with every edge annotated by the name of a transition rule that justifies the presence of this edge in the graph:

$$\begin{array}{lll} \emptyset & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta} & \Longrightarrow & (\textit{Unit Propagate}) \\ a^{\Delta}\ c & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta}\ c\ b^{\Delta} & & \end{array} \tag{3.1}$$

Since the state $a^{\Delta}\ c\ b^{\Delta}$ is terminal, Theorem 1(b) asserts that $\{a, c, b\}$ is a model of F. Here is another path in DP_F from $\emptyset$ to the same terminal node:

$$\begin{array}{lll} \emptyset & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta} & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta}\ \neg c^{\Delta} & \Longrightarrow & (\textit{Unit Propagate}) \\ a^{\Delta}\ \neg c^{\Delta}\ c & \Longrightarrow & (\textit{Backtrack}) \\ a^{\Delta}\ c & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta}\ c\ b^{\Delta} & & \end{array} \tag{3.2}$$

Path (3.1) corresponds to an execution of DPLL in the sense of [Davis *et al.*, 1962]; path (3.2) does not, because it applies *Decide* to a^{Δ} even though *Unit Propagate* could be applied in this state.

Note that the graph DP_F is a modification of the *classical DPLL* graph defined in [Nieuwenhuis *et al.*, 2006, Section 2.3]. It is different in three ways. First, its

states are pairs $M||F$ for all CNF formulas F. For the purposes of this section, it is not necessary to include F. Second, the description of the classical DPLL graph involves a "PureLiteral" transition rule. Third, in the definition of the graph in [Nieuwenhuis *et al.*, 2006, Section 2.3], each M is required to be consistent. In case of DPLL, due to the simple structure of a clause, it is possible to characterize the applicability of *Backtrack* in a simple manner: when some of the clauses become inconsistent with the current partial assignment, *Backtrack* is applicable. In ASP, it is not easy to describe the applicability of *Backtrack* if only consistent states are taken into account. We introduce inconsistent states in the graph DP_F, because in the Section 5.2 we will modify DP_F in order to characterize the computation of answer sets of a logic program by means of the SMODELS algorithm.

Theorem 1 is similar to Theorems 2.10 and 2.13 in [Nieuwenhuis *et al.*, 2006, Section 2.5] but they are not equivalent because the graphs considered in the theorems differ. We will present a proof of Theorem 1 in the rest of this section and we will refer to this proof later in the dissertation.

Lemma 1. *For any CNF formula F and any state $l_1 \ \ldots \ l_n$ reachable from $\emptyset$ in DP_F, every model X of F satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.*

Proof. By induction on the path from $\emptyset$ to $l_1 \ \ldots \ l_n$. The property of X that we need to prove trivially holds in the initial state $\emptyset$, and we will prove that all transition rules of DP_F preserve it.

Take a model X of F, and consider an edge $M \Longrightarrow M'$ where M is a list $l_1 \ \ldots \ l_k$ such that X satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.

It is clear that the rule justifying the transition from M to M' is different from *Fail*. For each of the other three rules, M' is obtained from a prefix of M by appending a list of literals containing at most one decision literal. Due to the inductive hypothesis, it is sufficient to show that if X satisfies all decision literals in M' then X satisfies all M'.

Unit Propagate: M' is $M \ l$. By the inductive hypothesis, for every literal in M the property in question holds. We need to show that $X \models l$. From the definition of *Unit Propagate*, for some clause $C \vee l \in F$, $\overline{C} \subseteq M$. Consequently, $M \models \neg C$. From the inductive hypothesis and the assumption that X satisfies all decision literals in M' and hence in M, it follows that $X \models M$. Since X is a model of F, we conclude that $X \models l$.

Decide: M' is $M\ l^{\Delta}$. Obvious.

Backtrack: M has the form $P\ l^{\Delta}\ Q$ where Q contains no decision literals. M' is $P\ \overline{l}$. By the inductive hypothesis, it trivially follows that for every literal in P the property in question holds. We need to show that $X \models \overline{l}$. Assume that $X \models l$. Since Q does not contain decision literals, and the assumption that X satisfies all decision literals in M' and hence in P, X satisfies all decision literals in $P\ l^{\Delta}\ Q$, that is M. By the inductive hypothesis, it follows that X satisfies M. This is impossible because M is inconsistent. □

Proof of Theorem 1

(a) The finiteness of DP_F is obvious. For any list N of literals by $|N|$ we denote the length of N. Any state M other than *FailState* has the form $M_0\ l_1^{\Delta}\ M_1 \ldots l_p^{\Delta}\ M_p$, where $l_1^{\Delta} \ldots l_p^{\Delta}$ are all decision literals of M; we define $\alpha(M)$ as the sequence of nonnegative integers $|M_0|, |M_1|, \ldots, |M_p|$, and $\alpha(\mathit{FailState}) = \infty$. By the definition of the transition rules defining the edges of DP_F, if there is an edge from a state M to M' in DP_F then $\alpha(M) < \alpha(M')$, where $<$ is understood as the lexicographical order. It follows that if a state M' is reachable from M then $\alpha(M) < \alpha(M')$. Consequently the graph is acyclic.

(b) Consider any terminal state M other than *FailState*. From the fact that *Decide* is not applicable, we conclude that M has no unassigned literals. Since neither *Backtrack* nor *Fail* is applicable, M is consistent. Consequently M is an assignment. It follows that for any clause $C \vee l \in F$ if $\overline{C} \not\subseteq M$ then $C \cap M \neq \emptyset$. Furthermore, since *Unit Propagate* is not applicable, we conclude that if $\overline{C} \subseteq M$ then $l \in M$. Consequently, $M \models C \vee l$. Hence M is a model of F.

(c) Left-to-right: Since *FailState* is reachable from $\emptyset$, there is an inconsistent state M without decision literals that is reachable from $\emptyset$. By Lemma 1, any model of F satisfies M. Since M is inconsistent we conclude that F has no models.

Right-to-left: From (a) it follows that there is a path from $\emptyset$ to some terminal state. By (b), this state cannot be different from *FailState*, because F is unsatisfiable. □

3.3 Strategies and Techniques

State-of-the-art SAT solvers augment the basic DPLL algorithm with a number of advanced features that make these systems applicable and successful in solving large SAT instances. In this section we provide a brief description of some of these sophisticated strategies.

Stallman and Sussman [1977] introduced *conflict-driven backjumping* that allows a solver to backtrack directly to a decision level where branching step took place on a variable that caused a conflict. This technique preserves the completeness of search but allows the solver to enhance its computation by skipping parts of the search tree that are not essential.

Clause *learning* [Marques-Silva and Sakallah, 1996b; Bayardo and Schrag, 1997; Zhang *et al.*, 2001; Dixon *et al.*, 2004] brought the field of propositional satisfiability to new computational heights. The main idea behind it is to learn, or in other words save, the so-called conflict clauses to the original database of clauses. Conflict clauses are gained from the preceding computation (once backtrack or backjump is performed), and help the solver to disregard the irrelevant search tree branches in the future. It has been shown that clause learning can exponentially improve the basic DPLL procedure [Beame *et al.*, 2004]. *Forgetting* is a technique that allows a solver to disregard previously learned clauses once they are not helpful. This technique permits controlling the growth of a solver's clause database.

Gomes et al. [1998] introduced the idea of randomized *restarts*, that allows a solver to interrupt its branching at one point of the tree and start the search over at another point. Baptista and Marques-Silva [2000] extended the approach further by combining it with the clause learning technique and permitting the solver to keep learned clauses as part of original clause database after a restart. Most of the modern SAT solvers employ restart strategies, sometimes restarting after as few as 20 to 50 backtracks. Once a solver restarts, it usually starts the search from scratch.

3.4 Abstract DPLL with Backjumping and Learning

Nieuwenhuis et al. [2006, Section 2.4] defined the *DPLL System with Learning* graph that can be used to describe most of the modern SAT solvers which typically implement such sophisticated techniques as backjumping, learning, forgetting, and

Unit Propagate λ:

$$M||\Gamma \implies M\ l||\Gamma \text{ if } \begin{cases} C \vee l \in F \cup \Gamma \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

Backjump:

$$P\ l^{\Delta}\ Q||\Gamma \implies P\ l'||\Gamma \text{ if } \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent and} \\ F \models l' \vee \overline{P} \end{cases}$$

Learn:

$$M||\Gamma \implies M||\ C,\ \Gamma \text{ if } \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ and} \\ F \models C \end{cases}$$

Figure 3.2: The additional transition rules of the graph DPLL_F.

restarts discussed in Section 3.3.

In this section we will extend the graph DP_F to capture the ideas behind backjumping and learning. The new graph will be closely related to the *DPLL System with Learning* graph introduced in [Nieuwenhuis *et al.*, 2006, Section 2.4].

We first note that the graph DP_F is not adequate to capture such technique as learning since it is incapable to reflect a change in a state of computation related to newly learned clauses. We start by redefining a state so that it incorporates information about changes performed on a clause database.

For a CNF formula F, an *augmented state* relative to F is either a distinguished state *FailState* or a pair $M||\Gamma$ where M is a record relative to the set of atoms occurring in F, and Γ is a (multi)set of clauses over atoms of F that are entailed by F.

We now define a graph DPLL_F for any CNF formula F. Its nodes are the augmented states relative to F. The transition rules *Decide* and *Fail* of DP_F are extended to DPLL_F as follows: $M||\Gamma \implies M'||\Gamma$ ($M||\Gamma \implies \mathit{FailState}$) is an edge in DPLL_F justified by *Decide* (*Fail*) if and only if $M \implies M'$ ($M \implies \mathit{FailState}$) is an edge in DP_F justified by *Decide* (*Fail*). Figure 3.2 presents the other transition rules of DPLL_F. We refer to the transition rules *Unit Propagate* λ, *Backjump*, *Decide*, and *Fail* of the graph DPLL_F as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn* is applicable to it.

We will omit the word "augmented" before "state" when this is clear from a context.

The graph DPLL_F can be used for deciding the satisfiability of a formula F

simply by constructing an arbitrary path from node $\emptyset||\emptyset$ to a semi-terminal node:

Theorem 2. *For any CNF formula F,*

(a) every path in DPL_F *contains only finitely many edges justified by Basic transition rules,*

(b) for any semi-terminal state $M||\Gamma$ of DPL_F *reachable from $\emptyset||\emptyset$, M is a model of F,*

(c) FailState is reachable from $\emptyset||\emptyset$ in DPL_F *if and only if F is unsatisfiable.*

On the one hand, Theorem 2 (a) asserts that if we construct a path from $\emptyset||\emptyset$ so that Basic transition rules periodically appear in it then some semi-terminal state will be eventually reached. On the other hand, Theorem 2 (b) and (c) assert that as soon as a semi-terminal state is reached the problem of deciding whether formula F is satisfiable is solved. The proof of this theorem is similar to the proof of Theorem 2.12 from [Nieuwenhuis *et al.*, 2006].

For instance, let F be the formula

$$\begin{array}{l} a \vee b \\ \neg a \vee c. \end{array}$$

Here is a path in DPL_F:

$$\begin{array}{lll} \emptyset||\emptyset & \Longrightarrow & (\textit{Learn}) \\ \emptyset||b \vee c & \Longrightarrow & (\textit{Decide}) \\ \neg b^{\Delta}||b \vee c & \Longrightarrow & (\textit{Unit Propagate } \lambda) \\ \neg b^{\Delta}\ c||b \vee c & \Longrightarrow & (\textit{Unit Propagate } \lambda) \\ \neg b^{\Delta}\ c\ a||b \vee c & & \end{array} \tag{3.3}$$

Since the state $\neg b^{\Delta}\ c\ a$ is semi-terminal, Theorem 2 (b) asserts that $\{\neg b, c, a\}$ is a model of F.

Recall that the transition rule *Backtrack* of the graph DP_F – a prototype of *Backjump* – is applicable in any inconsistent state with a decision literal in DP_F. The transition rule *Backjump*, on the other hand, is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$ (the proof of this statement is similar to the proof of Lemma 2.8 from [Nieuwenhuis *et al.*, 2006]). The application

of *Backjump* where l^{Δ} is the last decision literal and l' is $\overline{l}$ can be seen as an application of *Backtrack*. This fact shows that *Backjump* is essentially a generalization of *Backtrack*. The subgraph of DP_F induced by the nodes reachable from $\emptyset$ is basically a subgraph of DPL_F.

In order to model such techniques as forgetting and restarts Nieuwenhuis et al. [2006] extend the graph DPL_F with the following transition rules that capture the ideas behind these techniques:

Restart:
$$M||\Gamma \implies \emptyset||\Gamma$$

Forget:
$$M||C,\ \Gamma \implies M||\Gamma.$$

It is easy to prove a result similar to Theorem 2 for the graph DPL_F with *Restart* and *Forget* (for such graph a state is semi-terminal if no rule other than *Learn*, *Restart*, *Forget* is applicable to it.)

Chapter 4

Background: Traditional ASP Programs

This chapter starts by introducing logic programs with conventional Prolog syntax that we call traditional. Section 4.2 defines the answer set semantics, also called the stable model semantics, for such programs.

Our approach to computing answer sets heavily relies on the relation between the answer set semantics of logic programs and the completion semantics: beginning with Fages' findings that for "tight" logic programs these semantics coincide with each other. This fact led us to the idea of using SAT solvers as answer set solvers for tight programs, discussed in Chapter 6. Section 4.3 describes the concept of an unfounded set that is strongly related to an answer set. Section 4.4 introduces the concept of completion. Section 4.5 defines tightness and states the Fages theorem. Section 4.6 presents details on one of the best known answer set systems SMODELS. The system SMODELS consists of two major components: the grounder LPARSE and the solver SMODELS. Section 4.6 also describes the grounder GRINGO.

Although the syntax of logic programs allows variables, as for instance in *color.lp* program in Section 2.1, the theory and definitions that we provide here deal with propositional programs only. This is due to the fact that a rule with variables can be interpreted as an abbreviation for a set of propositional rules. In fact, most modern answer set solvers use off-the-shelf grounders, such as LPARSE, to perform a transformation of a program with variables into a propositional program. Therefore it is sufficient to review the definitions and theory for the propositional case.

4.1 Logic Program

A *traditional logic program* consists of *rules* of the form

$$a \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m \tag{4.1}$$

where a is a (propositional) atom or symbol $\bot$, and each b_i $(1 \leq i \leq m)$ is a (propositional) atom. We call such rules *traditional*. We call a the *head* of the rule, and

$$b_1, \ldots, b_l,\ not\ b_{l+1}, \ldots,\ not\ b_m$$

its *body*.

We will identify the body of (4.1) with the conjunction

$$b_1 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \ldots \neg b_m \tag{4.2}$$

and also with the set of its conjunctive terms. If the head a of a rule (4.1) is an atom then we will identify (4.1) with the clause

$$a \vee \neg b_1 \vee \cdots \vee \neg b_l \vee b_{l+1} \vee \ldots b_m. \tag{4.3}$$

If a is $\bot$ then we call rule (4.1) a *constraint* and identify (4.1) with the clause

$$\neg b_1 \vee \cdots \vee \neg b_l \vee b_{l+1} \vee \ldots b_m. \tag{4.4}$$

We will often use two abbreviated forms for a rule (4.1): The first is

$$a \leftarrow B \tag{4.5}$$

where B stands for $b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m$. The second abbreviation is

$$a \leftarrow D, F \tag{4.6}$$

where D stands for the *positive part of the body* $b_1, \ldots, b_l$, and F stands for the *negative part of the body* $not\ b_{l+1}, \ldots, not\ b_m$.

By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of Π with head a.

4.2 Answer Sets

In order to state the definition of an answer set (also called stable model) for a program we first need to define the notions of satisfaction and a reduct.

We identify a set of atoms with the truth assignment that maps the elements of the set to *True*, and all other atoms to *False*. The definition of when a set of atoms satisfies a rule, a head, or a body is the usual definition of satisfaction in propositional logic. We say that a set X of atoms $satisfies$ a program Π (symbolically, $X \models \Pi$) if X satisfies every rule of Π. We call such a set X a *model* of program Π.

The *reduct* Π^X of a program Π with respect to a set X of atoms is the program obtained from Π by

- removing each rule (4.6) such that $\overline{F} \cap X \neq \emptyset$, and
- replacing each remaining rule (4.6) by $a \leftarrow D$.

A set X of atoms is an *answer set* for a program Π if X is minimal (with respect to set inclusion) among the sets of atoms that satisfy the reduct Π^X [Gelfond and Lifschitz, 1988].

It is easy to show that an answer set M of a program Π is also always a model of Π.

For instance, let Π be the program

$$a \leftarrow not\ b$$
$$b \leftarrow not\ a.$$

Consider set $\{a\}$. Reduct $\Pi^{\{a\}}$ is

$$a \leftarrow \top$$
$$b \leftarrow \bot$$

or, equivalently,

$$a \leftarrow \tag{4.7}$$

Set $\{a\}$ satisfies the reduct and is minimal, hence $\{a\}$ is an answer set of Π.

Consider set $\{a, b\}$. The reduct $\Pi^{\{a,b\}}$ contains no rules. Hence $\emptyset$ is its minimal model. Set $\{a, b\}$ is not an answer set of Π.

Note that if some program Π consists only of the rules with the empty negative part F of the body, then Π is identical to the reduct of Π with respect to any set of atoms. Furthermore, if such a program Π does not contain constraints then Π can be seen as a set of Horn clauses (Horn formula). There is an algorithm linear in the size of the Horn formula for finding minimal model of the formula [Dowling and Gallier, 1984]. Trivially, this minimal model coincides with the unique answer set for such Π.

4.3 Unfounded Sets

For any set M of literals, by M^+ we denote the set of positive literals from M. For instance, $\{a,\ \neg b\}^+$ is $\{a\}$.

A set U of atoms occurring in a program Π is said to be *unfounded* [Van Gelder *et al.*, 1991] on a consistent set M of literals with respect to Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $M \models \neg B$ or $U \cap B^+ \neq \emptyset$. There is a tight relation between unfounded sets and answer sets: For any model M of a program Π, M^+ is an answer set for Π if and only if M contains no non-empty subsets unfounded on M with respect to Π (Corollary 2 from [Saccá and Zaniolo, 1990])[1].

For instance, let Π be the program

$$\begin{aligned} a &\leftarrow not\ b \\ b &\leftarrow not\ a \\ c &\leftarrow a \\ d &\leftarrow d. \end{aligned} \tag{4.8}$$

Let M be the consistent set $\{a, \neg b, c, d\}$ of literals. Set $M^+ = \{a, c, d\}$ is not an answer set of Π. Accordingly, its subset $\{d\}$ is unfounded on $\{a, \neg b, c, d\}$ with respect to Π, because the only rule in Π with d in the head

$$d \leftarrow d$$

is such that $U \cap B^+ = \{d\} \cap \{d\} \neq \emptyset$.

[1] Corollary 2 from [Saccá and Zaniolo, 1990] refers to "assumption sets" rather than unfounded sets. But, as the authors noted, in the context of this corollary the two concepts are equivalent.

4.4 Completion and Supported Models

Keith Clark [1978] proposed the reduction from logic programs to propositional formulas called completion. Originally, Clark introduced this notion for Prolog rules with variables. Within this dissertation, it is sufficient to state his definition for the case of propositional programs only.

The *completion* of program Π, $Comp(\Pi)$, can be defined as the set of propositional formulas that consists of the implications

$$B \rightarrow a \tag{4.9}$$

for all rules (4.5) in Π and the implications

$$a \rightarrow \bigvee_{B \in Bodies(\Pi, a)} B \tag{4.10}$$

for all atoms a occurring in Π.

The intuition behind formula (4.9) is that the bodies of the rules of Π with the head a can be viewed as sufficient conditions for a; (4.10) states that the disjunction of these sufficient conditions for a is also necessary.

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ a \\ c \leftarrow a \\ c \leftarrow b. \end{array} \tag{4.11}$$

Its completion is

$$\begin{array}{l} \neg b \rightarrow a \\ \neg a \rightarrow b \\ a \rightarrow c \\ b \rightarrow c \\ a \rightarrow \neg b \\ b \rightarrow \neg a \\ c \rightarrow a \vee b. \end{array}$$

Sets $\{a, c\}$ and $\{b, c\}$ are the only models of the completion.

For any program Π, models of its completion are also known as supported models of Π: a set M of atoms is a *supported* model of Π, if for every atom $a \in M$, $M \models B$ for some $B \in Bodies(\Pi, a)$. Indeed, for a program (4.11) its models of completion $\{a, c\}$ and $\{b, c\}$ are the only supported models of the program. Consider set $\{a, c\}$, it is "supported" by the first and the third rules of program (4.11).

4.5 Tightness

There is a close relation between the answer set semantics of logic programs and completion. For instance, Marek and Subrahmanian [1989] showed that any answer set of a program is also a model of its completion. Moreover, it is well know that the models of the program's completion coincide with the supported models of the program.

Theorem 3 (Theorem on Completion)**.** *[Marek and Subrahmanian, 1989] For any traditional program* Π*, any answer set for* Π *satisfies the program's completion* $Comp(\Pi)$*.*

Nevertheless, the converse does not always hold. Consider program (4.8). Its completion has four models:

$$\{a, c\} \quad \{b\} \quad \{a, c, d\} \quad \{b, d\}.$$

Only the first two models are answer sets of the program.

Fages [1994] defined a syntactic condition on a program called "tightness" such that if a logic program is "tight" then its answer sets are identical to the models of its completion.

The *dependency graph* of a program Π is the directed graph G such that

- the vertices of G are the atoms occurring in Π,
- for every rule (4.6) in Π that is not a constraint, G has an edge from atom a to each atom in D .

A program is called *tight* if its dependency graph is acyclic.

For instance, consider program (4.11). Its dependency graph, shown in Figure 4.1, is acyclic. Hence the program is tight.

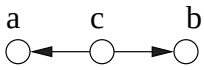

Figure 4.1: Dependency graph of program (4.11)

On the other hand, the dependency graph for program (4.8) contains a cycle from vertex d to itself. Program (4.8) is not tight.

Fages [1994] proved that for any tight program, its answer sets and models of its completion coincide.

Theorem 4 (Theorem on Tight Programs)**.** *For any tight traditional program* Π *and any set* X *of atoms,* X *is an answer set for* Π *if and only if* X *satisfies* $Comp(\Pi)$*.*

For instance, consider the tight program (4.11). The sets $\{a, c\}$, $\{b, c\}$ are the only answer sets of the program, and they are identical to the models of the program's completion.

Fages' observation led us to the idea that for tight programs SAT solvers can be used for computing answer sets of logic programs [Babovich *et al.*, 2000].

4.6 Answer Set Solver Smodels, and Grounders Lparse and Gringo

The answer set programming system SMODELS consists of two components: the grounder LPARSE and the solver SMODELS. The answer set solver CMODELS, which is our implementation of the SAT-based method introduced in this dissertation, utilizes the grounder LPARSE of SMODELS as its front-end.

SMODELS, with its front-end LPARSE, is one of the best known answer set systems. Its programs are composed of atoms and logic rules. In this chapter we only cover the case of traditional rules, but the syntax of SMODELS is more general. Chapter 7 will talk about programs containing rules with more general syntax. The front-end LPARSE performs the "grounding" on the program by instantiating its variables.

The description of the grounding procedure that LPARSE performs is beyond the scope of this work (see [Syrjanen, 2003]). But here we provide an example to demonstrate the ideas behind LPARSE. Consider the following logic program *test.lp*:

```
a(1).
a(2).
b(X):-a(X), not c(X).
c(X):-a(X), not b(X).
```

The grounder LPARSE determines two possible values, i.e., 1 and 2, for the variable X, because in each rule where X occurs, X also appears in the predicate a in the positive part of the body, which holds for 1 and 2. LPARSE produces the following traditional program:

```
a(1).
a(2).
b(1) :- not c(1).
b(2) :- not c(2).
c(1) :- not b(1).
c(2) :- not b(2).
```

Note that LPARSE also simplifies a program while grounding. For instance, predicate a disappeared from bodies of the rules in the program.

The command line

```
% lparse test.lp | smodels
```

invokes the grounder LPARSE and the answer set solver SMODELS on this sample program. The output of the system is

```
smodels version 2.34. Reading...done
Answer: 1
Stable Model: c(2) b(1) a(2) a(1)
```

This is only one of four answer sets of the program. The command line

```
% lparse test.lp | smodels 0
```

would instruct SMODELS to find all answer sets.

In [Gebser *et al.*, 2007a], the authors introduced a grounder, called GRINGO, which input language features a wider class of programs than accepted by LPARSE. Its output language coincides with the one of LPARSE. GRINGO supports a number of

sophisticated techniques that often allow it to produce more concise ground instances of problems than LPARSE. Similarly to the use of LPARSE in combination with SMODELS, the command line

```
% gringo test.lp | smodels
```

invokes the grounder GRINGO and the answer set solver SMODELS on the sample program *test.lp*.

The answer set solver SMODELS has been used for solving search problems in many domains. It could successfully compete with a special purpose commercial integer programming tool CPLEX in a verification application[2] [Heljanko, 1999]. Similarly, Heljanko and Niemelä [2003] demonstrated that SMODELS could be efficiently applied to bounded LTL model checking. Also SMODELS was used as part of a multi-platform toolchain TOAST [Brain *et al.*, 2006] that applies super optimization techniques to machine instruction programs. TOAST uses answer set programming as a scalable computational engine for conducting search over complex, non-regular domains.

The computational ideas behind the answer set solver SMODELS are closely related to the classical SAT procedure DPLL. In the next chapter we will introduce the SMODELS algorithm by means of an abstract framework similar to the one in Section 3.2.

[2]Mcsmodels tool http://www.tcs.hut.fi/∼kepa/tools/

Chapter 5

Abstract Description of Answer Set Solvers

In this chapter we provide details on the answer set solver SMODELS by describing its algorithm using an abstract framework similar to the one introduced in Section 3.2. In Section 5.2 we define a graph representing the application of the SMODELS algorithm to a program, and in Section 5.3 we describe this algorithm by means of the graph. As a step in this direction, in Section 5.1 we describe a simpler graph representing an algorithm for generating supported models of a program. Section 5.1.2 talks about the relation between DPLL and an algorithm for generating supported models of a program. Section 5.4 establishes the tight relation between the SMODELS procedure applied to a tight program and DPLL applied to the program's completion.

5.1 Generating Supported Models

In Section 5.2 we will define, for an arbitrary program Π, a graph SM_Π representing the application of the SMODELS algorithm to Π; the terminal nodes of SM_Π are answer sets of Π. We start by describing a simpler graph ATLEAST_Π.

5.1.1 Graph ATLEAST_Π

The terminal nodes of ATLEAST_Π are supported models of Π[1].

[1]The transition rules defining ATLEAST_Π are closely related to procedure *Atleast* [Simons, 2000, Sections 4.1], which is one of the core procedures of the SMODELS algorithm.

Unit Propagate LP:

$$M \implies M\ a \text{ if } \begin{cases} a \leftarrow B \in \Pi \text{ and} \\ B \subseteq M \end{cases}$$

All Rules Cancelled:

$$M \implies M\ \neg a \text{ if } \overline{B} \cap M \neq \emptyset \text{ for all } B \in Bodies(\Pi, a)$$

Backchain True:

$$M \implies M\ l \text{ if } \begin{cases} a \leftarrow B \in \Pi, \\ a \in M, \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in Bodies(\Pi, a) \setminus B\ , \\ l \in B \end{cases}$$

Backchain False:

$$M \implies M\ \overline{l} \text{ if } \begin{cases} a \leftarrow l, B\ \in\ \Pi, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M \end{cases}$$

Figure 5.1: The additional transition rules of the graph ATLEAST$_\Pi$.

The nodes of ATLEAST$_\Pi$ are the states relative to the set of atoms occurring in Π. The edges of the graph ATLEAST$_\Pi$ are described by the transition rules *Decide*, *Fail*, *Backtrack* introduced in Section 3.2 and the additional transition rules[2] presented in Figure 5.1. Note that each of the rules *Unit Propagate LP* and *Backchain False* is similar to *Unit Propagate*: the former corresponds to *Unit Propagate* on $C \vee l$ where l is the head of the rule, and the latter corresponds to *Unit Propagate* on $C \vee l$ where $\overline{l}$ is an element of the body of the rule.

This graph can be used for deciding whether program Π has a supported model by constructing a path from $\emptyset$ to a terminal node:

Theorem 5. *For any program* Π*,*

(a) graph ATLEAST$_\Pi$ *is finite and acyclic,*

(b) any terminal state of ATLEAST$_\Pi$ *other than FailState is a supported model of* Π*,*

(c) FailState is reachable from $\emptyset$ *in* ATLEAST$_\Pi$ *if and only if* Π *has no supported models.*

[2]The names of some of these rules follow [Ward, 2004].

For instance, let Π be program (4.8). Here is a path in ATLEAST_Π:

$$\begin{array}{lll} \emptyset & \Longrightarrow & (\textit{Decide}) \\ a^\Delta & \Longrightarrow & (\textit{Unit Propagate LP}) \\ a^\Delta\ c & \Longrightarrow & (\textit{All Rules Cancelled}) \\ a^\Delta\ c\ \neg b & \Longrightarrow & (\textit{Decide}) \\ a^\Delta\ c\ \neg b\ d^\Delta & & \end{array} \tag{5.1}$$

Since the state $a^\Delta\ c\ \neg b\ d^\Delta$ is terminal, Theorem 5(b) asserts that $\{a, c, \neg b, d\}$ is a supported model of Π.

The assertion of Theorem 5 will remain true if we drop the transition rules *Backchain True* and *Backchain False* from the definition of ATLEAST_Π.

In the rest of this section we give a proof of Theorem 5.

Lemma 2. *For any program* Π *and any state* $l_1\ \ldots\ l_n$ *reachable from* $\emptyset$ *in* ATLEAST_Π, *every supported model* X *for* Π *satisfies* l_i *if it satisfies all decision literals* l_j^Δ *with* $j \leq i$.

Proof. By induction on the path from $\emptyset$ to $l_1\ \ldots\ l_n$. Similar to the proof of Lemma 1. We will show that the property in question is preserved when the transition from M to M' is justified by any of the four new rules.

Take a supported model X for Π, and consider an edge $M \Longrightarrow M'$ where M is a list $l_1\ \ldots\ l_k$ such that X satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.

Assume that X satisfies all decision literals in M'.

Unit Propagate LP: M' is $M\ a$. By the inductive hypothesis, for every literal in M the property in question holds. We need to show that $X \models a$. By the definition of *Unit Propagate LP*, $B \subseteq M$ for some rule $a \leftarrow B$. Consequently, $M \models B$. From the inductive hypothesis and the assumption that X satisfies all decision literals in M' and hence in M, it follows that $X \models M$. Since X is a model of Π we conclude that $X \models a$.

All Rules Cancelled: M' is $M\ \neg a$ and $\overline{B} \cap M \neq \emptyset$ for every $B \in \mathit{Bodies}(\Pi, a)$. Consequently, $M \models \neg B$ for every $B \in \mathit{Bodies}(\Pi, a)$. By the inductive hypothesis, for every literal in M the property in question holds. We need to show that $X \models \neg a$. By contradiction. Assume that $X \models a$. From the inductive hypothesis and the

assumption that X satisfies all decision literals in M' and hence in M, it follows that $X \models M$. Since $M \models \neg B$ for every $B \in Bodies(\Pi, a)$, it follows that $X \models \neg B$. We conclude that X is not a supported model of Π.

Backchain True: M' is $M\ l$. By the inductive hypothesis, for every literal in M the property in question holds. We need to show that $X \models l$. By contradiction. Assume $X \models \overline{l}$. Consider the rule $a \leftarrow B$ corresponding to this application of *Backchain True*. Since $l \in B$, $X \models \neg B$. By the definition of *Backchain True*, $\overline{B'} \cap M \neq \emptyset$ for every B' in $Bodies(\Pi, a) \setminus B$. Consequently, $M \models \neg B'$ for every B' in $Bodies(\Pi, a) \setminus B$. From the inductive hypothesis and the assumption that X satisfies all decision literals in M' and hence in M, it follows that $X \models M$. We conclude that $X \models \neg B'$ for every B' in $Bodies(\Pi, a) \setminus B$. Hence X is not supported by Π.

Backchain False: M' is $M\ \overline{l}$. By the inductive hypothesis, for every literal in M the property in question holds. We need to show that $X \models \overline{l}$. By contradiction. Assume that $X \models l$. By the definition of *Backchain False* there exists a rule $a \leftarrow l, B$ in Π such that $\neg a \in M$ and $B \subseteq M$. Consequently, $M \models \neg a$ and $M \models B$. From the inductive hypothesis and the assumption that X satisfies all decision literals in M' and hence in M, it follows that $X \models M$. We conclude that $X \models \neg a$ and $X \models B$. From the fact that $X \models l$, it follows that X does not satisfy the rule $a \leftarrow l, B$, so that it is not a model of Π. □

Proof of Theorem 5

Parts (a) and (c) are proved as in the proof of Theorem 1, using Lemma 2.

(b) Let M be a terminal state so that none of the rules are applicable. From the fact that *Decide* is not applicable, we conclude that M assigns all literals. Since neither *Backtrack* nor *Fail* is applicable, M is consistent. Consequently, M is an assignment. Since *Unit Propagate LP* is not applicable, it follows that for every rule $a \leftarrow B \in \Pi$, if $B \subseteq M$ then $a \in M$. Consequently, if $M \models B$ then $M \models a$. We conclude that M is a model of Π. We will now show that M is a supported model of Π. By contradiction. Suppose that M is not a supported model. Then, there is an atom $a \in M$ such that $M \not\models B$ for every $B \in Bodies(\Pi, a)$. Since M is consistent, $\overline{B} \cap M \neq \emptyset$ for every $B \in Bodies(\Pi, a)$. Consequently, *All Rules Cancelled* is applicable. This contradicts the assumption that M is terminal. □

The fact that the assertion of Theorem 5 remains true if we drop the tran-

sition rules *Backchain True* and *Backchain False* from the definition of ATLEAST_Π follows from the proof of Theorem 5 (b) that does not refer to those rules.

5.1.2 Relation between DP_F and ATLEAST_Π

Recall that the supported models of a program can be characterized as models of the program's completion. It turns out that the graph ATLEAST_Π is identical to the graph DP_F, where F is the (clausified) completion of Π. To make this claim precise, we first review the notion of completion.

For any program Π, its completion consists of Π and the formulas that can be written as

$$\neg a \vee \bigvee_{B \in Bodies(\Pi, a)} B \tag{5.2}$$

for every atom a in Π. *CNF-Comp*(Π) is the completion converted to CNF using straightforward equivalent transformations. In other words, *CNF-Comp*(Π) consists of clauses of two kinds:

1. the rules $a \leftarrow B$ of the program written as clauses

$$a \vee \overline{B}, \tag{5.3}$$

2. formulas (5.2) converted to CNF using the distributivity of disjunction over conjunction[3].

Theorem 6. *For any program* Π*, the graphs* ATLEAST_Π *and* $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ *are equal.*

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow b,\ not\ c \\ b. \end{array} \tag{5.4}$$

Its completion is

$$(a \leftrightarrow b \wedge \neg c) \wedge b \wedge \neg c, \tag{5.5}$$

[3]It is essential that repetitions are not removed in the process of clausification. For instance, *CNF-Comp*($a \leftarrow not\ a$) is the formula $(a \vee a) \wedge (\neg a \vee \neg a)$.

and *CNF-Comp*(Π) is

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg c) \wedge b \wedge \neg c. \tag{5.6}$$

Theorem 6 asserts that ATLEAST_Π coincides with $\text{DP}_{\textit{CNF-Comp}(\Pi)}$.

From Theorem 6 it follows that applying the *Atleast* algorithm to a program essentially amounts to applying DPLL to its completion.

In the rest of this section we give a proof of Theorem 6.

It is easy to see that the states of the graphs ATLEAST_Π and $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ coincide. We will now show that the edges of ATLEAST_Π and $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ coincide also.

It is clear that there is an edge $M \implies M'$ in ATLEAST_Π justified by the rule *Decide* if and only if there is an edge $M \implies M'$ in $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ justified by *Decide*. The same holds for the transition rules *Fail* and *Backtrack*.

We will now show that if there is an edge from a state M to a state M' in the graph $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ justified by the transition rule *Unit Propagate* then there is an edge from M to M' in ATLEAST_Π. Consider a clause $C \vee l \in \textit{CNF-Comp}(\Pi)$ such that $\overline{C} \subseteq M$. We will consider two cases, depending on whether $C \vee l$ comes from (5.3) or from the CNF of (5.2).

Case 1. $C \vee l$ is $a \vee \overline{B}$ corresponding to a rule $a \leftarrow B$.

Case 1.1. l is a. Then there is an edge from M to M' in ATLEAST_Π justified by the transition rule *Unit Propagate LP*.

Case 1.2. l is an element of $\overline{B}$. Then B has the form $\overline{l}, D$ and C is $a \vee \overline{D}$. From $\overline{C} \subseteq M$ we conclude that $D \subseteq M$ and $\neg a \in M$. There is an edge from M to M' in the graph ATLEAST_Π justified by the following instance of *Backchain False*:

$$M \implies M\ l \ \text{ if } \begin{cases} a \leftarrow \overline{l}, D \ \in \ \Pi, \\ \neg a \in M, \\ D \subseteq M. \end{cases}$$

Case 2. $C \vee l$ has the form $\neg a \vee D$, where D is one of the clauses of the CNF of

$$\bigvee_{B \in \textit{Bodies}(\Pi, a)} B.$$

Then D has the form

$$\bigvee_{B \in Bodies(\Pi, a)} f(B)$$

where f is a function that maps every $B \in Bodies(\Pi, a)$ to an element of B.

Case 2.1. l is $\neg a$. Then C is D, so that $\overline{D} \subseteq M$. Consequently $\overline{f(B)} \in \overline{B} \cap \overline{D} \subseteq \overline{B} \cap M$, so that $\overline{B} \cap M \neq \emptyset$ for every $B \in Bodies(\Pi, a)$. There is an edge from M to M' in ATLEAST_Π justified by *All Rules Cancelled*.

Case 2.2. l is an element of D. From the construction of D, it follows that $l = f(B) \in B$ for some rule $a \leftarrow B$. Then C is

$$\neg a \vee \bigvee_{B' \in Bodies(\Pi, a) \setminus B} f(B').$$

From $\overline{C} \subseteq M$ we conclude that $a \in M$ and that $\overline{f(B')} \in M$ for every $B' \in Bodies(\Pi, a) \setminus B$. Since $f(B')$ is a conjunctive term of B', it follows that $\overline{B'} \cap M \neq \emptyset$. Then there is an edge from M to M' in ATLEAST_Π justified by *Backchain True*.

We will now show that if there is an edge from a state M to a state M' in the graph ATLEAST_Π justified by one of the transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, and *Backchain False* then there is an edge from M to M' in $\text{DP}_{\textit{CNF-Comp}(\Pi)}$.

Case 1. The edge is justified by *Unit Propagate LP*. Then there is a rule $a \leftarrow B \in \Pi$ where $B \subseteq M$, and M' is $M\ a$. By the construction of *CNF-Comp*(Π), $a \vee \overline{B} \in$ *CNF-Comp*(Π). There is an edge from M to M' in $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ justified by the following instance of *Unit Propagate*:

$$M \implies M\ a \text{ if } \begin{cases} \overline{B} \vee a \in \textit{CNF-Comp}(\Pi) \text{ and} \\ B \subseteq M. \end{cases}$$

Case 2. The edge is justified by *All Rules Cancelled*. By the definition of *All Rules Cancelled*, there is an atom a such that for all $B \in Bodies(\Pi, a)$, $\overline{B} \cap M \neq \emptyset$; and M' is $M\ \neg a$. Consequently, M contains the complement of some literal in B. Denote that literal by $f(B)$, so that $\overline{f(B)} \in M$. From the construction of *CNF-Comp*(Π),

$$\neg a \vee \bigvee_{B \in Bodies(\Pi, a)} f(B)$$

belongs to *CNF-Comp*(Π). By the choice of f,

$$\overline{\bigvee_{B \in Bodies(\Pi,a)} f(B)} \subseteq M.$$

There is an edge from M to M' in $\mathrm{DP}_{\mathit{CNF\text{-}Comp}(\Pi)}$ justified by the following instance of *Unit Propagate*:

$$M \implies M\ \neg a \text{ if } \begin{cases} \bigvee_{B \in Bodies(\Pi,a)} f(B) \vee \neg a \in \mathit{CNF\text{-}Comp}(\Pi), \\ \overline{\bigvee_{B \in Bodies(\Pi,a)} f(B)} \subseteq M. \end{cases}$$

Case 3. The edge is justified by *Backchain True*. By the definition of *Backchain True*, there is a rule $a \leftarrow B \in \Pi$ and a literal l such that $a \in M$; for all $B' \in Bodies(\Pi, a) \setminus B$, $\overline{B'} \cap M \neq \emptyset$; $l \in B$; and M' is $M\ l$. Let $f(B')$ be an element of B' such that $\overline{f(B')} \in M$. From the construction of *CNF-Comp*(Π),

$$\neg a \vee l \vee \bigvee_{B' \in Bodies(\Pi,a) \setminus B} f(B')$$

belongs to *CNF-Comp*(Π). By the choice of f,

$$\overline{\bigvee_{B' \in Bodies(\Pi,a) \setminus B} f(B')} \subseteq M.$$

There is an edge from M to M' in $\mathrm{DP}_{\mathit{CNF\text{-}Comp}(\Pi)}$ justified by the following instance of *Unit Propagate*:

$$M \implies M\ l \text{ if } \begin{cases} \neg a \vee l \vee \bigvee_{B' \in Bodies(\Pi,a) \setminus B} f(B') \in \mathit{CNF\text{-}Comp}(\Pi), \\ \overline{(\neg a \vee \bigvee_{B' \in Bodies(\Pi,a) \setminus B} f(B'))} \subseteq M. \end{cases}$$

Case 4. The edge is justified by *Backchain False*. By the definition of *Backchain False*, there is a rule $a \leftarrow l, B \in \Pi$ such that $\neg a \in M$, $B \subseteq M$, and M'

is $M\ \overline{l}$. By the construction of *CNF-Comp*(Π), $a \vee \overline{B} \vee \overline{l} \in$ *CNF-Comp*(Π). There is an edge from M to M' in $\mathrm{DP}_{\textit{CNF-Comp}(\Pi)}$ justified by the following instance of *Unit Propagate*:

$$M \implies M\ \overline{l} \text{ if } \begin{cases} a \vee \overline{B} \vee \overline{l} \in \textit{CNF-Comp}(\Pi) \text{ and} \\ \overline{a \vee \overline{B}} \subseteq M. \end{cases}$$

5.2 Abstract Smodels

We now describe the graph SM_Π that represents the application of the SMODELS algorithm to program Π. The nodes of SM_Π are the same as the nodes of the graph $\mathrm{ATLEAST}_\Pi$. The edges of SM_Π are described by the transition rules of $\mathrm{ATLEAST}_\Pi$ and the additional transition rule:

Unfounded:

$$M \implies M\ \neg a \text{ if } \begin{cases} M \text{ is consistent, and} \\ a \in U \text{ for a set } U \text{ unfounded on } M \text{ with respect to } \Pi. \end{cases}$$

This transition rule of SM_Π is closely related to procedure *Atmost* [Simons, 2000, Sections 4.2], which together with the procedure *Atleast* forms the core of the SMODELS algorithm.

The graph SM_Π can be used for deciding whether program Π has an answer set by constructing a path from $\emptyset$ to a terminal node:

Theorem 7. *For any program* Π*,*

(a) graph SM_Π *is finite and acyclic,*

(b) for any terminal state M *of* SM_Π *other than FailState,* M^+ *is an answer set of* Π*,*

(c) FailState is reachable from $\emptyset$ *in* SM_Π *if and only if* Π *has no answer sets.*

To illustrate the difference between SM_Π and $\mathrm{ATLEAST}_\Pi$, assume again that Π is program (4.8). Path (5.1) in the graph $\mathrm{ATLEAST}_\Pi$ is also a path in SM_Π. But state $a^\Delta\ c\ \neg b\ d^\Delta$, which is terminal in $\mathrm{ATLEAST}_\Pi$, is not terminal in SM_Π. This is not surprising, since $\{a, c, \neg b, d\}^+ = \{a, c, d\}$ is not an answer set of Π. To get to a

state that is terminal in SM_Π, we need two more steps:

$$\begin{array}{lll} \vdots & & \\ a^\Delta\ c\ \neg b\ d^\Delta & \Longrightarrow & (\mathit{Unfounded},\ U = \{d\}) \\ a^\Delta\ c\ \neg b\ d^\Delta\ \neg d & \Longrightarrow & (\mathit{Backtrack}) \\ a^\Delta\ c\ \neg b\ \neg d & & \end{array} \tag{5.7}$$

Theorem 7(b) asserts that $\{a, c\}$ is an answer set of Π.

The assertion of Theorem 7 will remain true if we drop the transition rules *All Rules Cancelled*, *Backchain True*, and *Backchain False* from the definition of SM_Π.

In the rest of this section we give a proof of Theorem 7.

We say that a model M of a program Π is *unfounded-free* if no non-empty subset of M is an unfounded set on M with respect to Π.

Lemma 3 (Corollary 2 from [Saccá and Zaniolo, 1990])**.** *For any model M of a program Π, M^+ is an answer set for Π if and only if M is unfounded-free.*

Lemma 4. *For any unfounded set U on a consistent set M of literals with respect to a program Π, and any assignment X, if $X \models M$ and $X \cap U \neq \emptyset$, then X^+ is not an answer set for Π.*

Proof. Assume that X^+ is an answer set for Π. Then X is a model of Π. By Lemma 3, it follows that X is unfounded-free. Hence any non-empty subset of X including $X \cap U$ is not unfounded on X. This means that for some rule $a \leftarrow B$ in Π such that $a \in X \cap U$, $\overline{B} \cap X = \emptyset$ and $X \cap U \cap B^+ = \emptyset$. From $X \models M$ ($M \subseteq X$) and $\overline{B} \cap X = \emptyset$ we conclude that $\overline{B} \cap M = \emptyset$. Since $\overline{B} \cap X = \emptyset$ and X is an assignment, $B \subseteq X$. It follows that $B^+ \subseteq X$. Consequently $U \cap B^+ = X \cap U \cap B^+ = \emptyset$. This contradicts the assumption that U is an unfounded set on M. □

Lemma 5. *For any program Π, any state $l_1\ \ldots\ l_n$ reachable from $\emptyset$ in SM_Π, and any assignment X, if X^+ is an answer set for Π then X satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.*

Proof. By induction on the path from $\emptyset$ to $l_1\ \ldots\ l_n$. Recall that for any assignment X, if X^+ is an answer set for Π, then X is a supported model of Π, and that the transition system SM_Π extends ATLEAST_Π only by the transition rule *Unfounded*.

Given our proof of Lemma 2, we only need to demonstrate that application of *Unfounded* preserves the property.

Consider a transition $M \Longrightarrow M'$ justified by *Unfounded*, where M is a sequence $l_1 \ \ldots \ l_k$. M' is $M \ \neg a$, such that $a \in U$, where U is an unfounded set on M with respect to Π. Take any assignment X such that X^+ is an answer set for Π and X satisfies all decision literals l_j^Δ with $j \leq k$. By the inductive hypothesis, $X \models M$. Then $X \models \neg a$. Indeed, otherwise a would be a common element of X and U, and $X \cap U$ would be non-empty, which contradicts Lemma 4. □

Proof of Theorem 7
Parts (a) and (c) are proved as in the proof of Theorem 1, using Lemma 5.
(b) As in the proof of Theorem 5(b) we conclude that M is a model of Π. Assume that M^+ is not an answer set. Then, by Lemma 3, there is a non-empty unfounded set U on M with respect to Π such that $U \subseteq M$. It follows that *Unfounded* is applicable (with an arbitrary $a \in U$). This contradicts the assumption that M is terminal. □

The fact that the assertion of Theorem 7 remains true if we drop the transition rules *All Rules Cancelled*, *Backchain True*, and *Backchain False* from the definition of SM_Π follows from the proof of Theorem 7 (b) that does not refer to those rules.

5.3 Smodels Algorithm

We can view a path in the graph SM_Π as a description of a process of search for an answer set for a program Π by applying inference rules. Therefore, we can characterize the algorithm of an answer set solver that utilizes the inference rules of SM_Π by describing a strategy for choosing a path in SM_Π. A strategy can be based, in particular, on assigning priorities to some or all inference rules of SM_Π, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state.

We use this method to describe the SMODELS algorithm. The system SMOD-

ELS assigns priorities to the inference rules of SM_Π as follows:

Backtrack, *Fail* >>
Unit Propagate LP, *All Rules Cancelled*, *Backchain True*, *Backchain False* >>
Unfounded >>
Decide.

For example, let Π be program (4.8). The SMODELS algorithm may follow a path

$$\begin{array}{lll} \emptyset & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta} & \Longrightarrow & (\textit{Unit Propagate LP}) \\ a^{\Delta}\ c & \Longrightarrow & (\textit{All Rules Cancelled}) \\ a^{\Delta}\ c\ \neg b & \Longrightarrow & (\textit{Unfounded}) \\ a^{\Delta}\ c\ \neg b\ \neg d & & \end{array}$$

in the graph SM_Π, whereas it may never follow path (5.1), because *Unfounded* has a higher priority than *Decide*.

5.4 Tight Programs: Smodels and DPLL

Recall that for any program Π and any assignment M, if M^+ is an answer set of Π then M is a supported model of Π. For the case of tight programs, the converse holds also: M^+ is an answer set for Π if and only if M is a supported model of Π [Fages, 1994] or, in other words, is a model of the completion of Π.

It turns out that for tight programs the graph SM_Π is "almost identical" to the graph DP_F, where F is the clausified completion of Π. To make this claim precise, we need the following terminology.

We say that an edge $M \implies M'$ in the graph SM_Π is *singular* if

- the only transition rule justifying this edge is *Unfounded*, and
- some edge $M \implies M''$ can be justified by a transition rule other than *Unfounded* or *Decide*.

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow\ b \\ b \leftarrow\ c. \end{array}$$

The edge

$$\begin{array}{ll} a^{\Delta}\ b^{\Delta}\ \neg c^{\Delta} & \Longrightarrow\ (\mathit{Unfounded},\ U = \{a, b\}) \\ a^{\Delta}\ b^{\Delta}\ \neg c^{\Delta}\ \neg a & \end{array}$$

in the graph SM_Π is singular, because the edge

$$\begin{array}{ll} a^{\Delta}\ b^{\Delta}\ \neg c^{\Delta} & \Longrightarrow\ (\mathit{All\ Rules\ Cancelled}) \\ a^{\Delta}\ b^{\Delta}\ \neg c^{\Delta}\ \neg b & \end{array}$$

belongs to SM_Π also.

With respect to the actual SMODELS algorithm [Simons, 2000], singular edges of the graph SM_Π are inessential: in view of priorities for choosing a path in SM_Π described in Section 5.3, SMODELS never follows a singular edge. Indeed, the transition rule *Unfounded* has the lower priority than any other transition rule but *Decide*. By SM^-_Π we denote the graph obtained from SM_Π by removing all singular edges.

Theorem 8. *For any tight program* Π*, the graph* SM^-_Π *is equal to each of the graphs* ATLEAST_Π *and* $\text{DP}_{\mathit{CNF\text{-}Comp}(\Pi)}$.

For instance, let Π be the program (5.4). This program is tight, its completion is (5.5), and *CNF-Comp*(Π) is formula (5.6). Theorem 8 asserts that SM^-_Π coincides with $\text{DP}_{\mathit{CNF\text{-}Comp}(\Pi)}$ and with ATLEAST_Π.

From Theorem 8, it follows that applying the SMODELS algorithm to a tight program essentially amounts to applying DPLL to its completion. A similar relationship, in terms of pseudocode representations of SMODELS and DPLL, is established in [Giunchiglia and Maratea, 2005].

In the rest of this section we give a proof of Theorem 8.

Lemma 6. *For any tight program* Π *and any non-empty unfounded set* U *on a consistent set* M *of literals with respect to* Π *there is an atom* $a \in U$ *such that for every* $B \in \mathit{Bodies}(\Pi, a)$, $\overline{B} \cap M \neq \emptyset$.

Proof. By contradiction. Assume that, for every $a \in U$ there exists $B \in \mathit{Bodies}(\Pi, a)$ such that $\overline{B} \cap M = \emptyset$. By the definition of an unfounded set it follows that for every atom $a \in U$ there is $B \in \mathit{Bodies}(\Pi, a)$ such that $U \cap B^+ \neq \emptyset$. Consequently the subgraph of the positive dependency graph of Π induced by U has no terminal nodes. Then, the program Π is not tight. □

Proof of Theorem 8

In view of Theorem 6, it is sufficient to prove that SM^-_Π equals ATLEAST_Π; or, in other words, that every edge of SM_Π justified by the rule *Unfounded* only is singular. Consider such an edge $M \implies M'$. We need to show that some transition rule other than *Unfounded* or *Decide* is applicable to M. By the definition of *Unfounded*, M is consistent and there exists a non-empty set U unfounded on M with respect to Π. By Lemma 6, it follows that there is an atom $a \in U$ such that for every $B \in \mathit{Bodies}(\Pi, a)$, $\overline{B} \cap M \neq \emptyset$. Therefore, the transition rule *All Rules Cancelled* is applicable to M. □

Chapter 6

Cmodels Algorithm for Tight Programs

Programs used in answer set programming, including those related to planning and commonsense reasoning, are often tight. As discussed in previous sections, Fages' result that the answer sets of a tight program coincide with the models of its completion suggests the possibility of using satisfiability solvers for finding answer sets of tight programs.

In this chapter we identify procedures necessary for creating a SAT-based answer set system. As mentioned in introductory Chapter 1, the answer set solver CMODELS is an implementation of the ideas described in this dissertation. Like SMODELS, it utilizes the grounder LPARSE (or GRINGO) as its front-end to obtain propositional logic program. Here we outline the CMODELS algorithm for finding answer sets of tight programs. The system performs the following steps during its execution:

- simplifies the program,
- verifies its tightness,
- produces its completion,
- clausifies the completion,
- invokes a SAT solver, and

- interprets its output.

The system CMODELS provides an interface to four SAT solvers MINISAT[1], RELSAT[2], SIMO[3], and ZCHAFF[4]. All of these SAT solvers implement such sophisticated features as backjumping, learning, and forgetting. It follows that the underlying search algorithm used by CMODELS on tight programs can be characterized by the graph DPL_F (see Section 3.4) where F is the clausified completion of a program.

This chapter covers details of the CMODELS algorithm. Section 6.1 covers the theory behind reduction techniques that allow us to reduce the size of the program and hence the search space. Section 6.2 talks about a possible way to verify the tightness of a program. Section 6.3 introduces a procedure for producing the clausified completion of a program. Section 6.4 compares the CMODELS approach to SMODELS and describes the experimental results that demonstrate the viability of our approach to computing answer sets in the case of tight programs.

6.1 Simplifying Traditional Programs

This section presents two theorems that provide the grounds for possible program simplification procedures. Although these results were previously known and used in the SMODELS implementation [Simons, 2000], they have not been explicitly stated. It is important for us to do this, because in Section 8.2 we will introduce the generalizations of these theorems to programs with more general syntax.

Often after grounding performed by LPARSE or GRINGO a program may include spurious rules or parts of rules that can be eliminated by the simplification procedures based on the theory presented here. Such simplifications will reduce the size of a given program and consequently the size of clausified completion and the search space.

For instance, consider a program

```
d(1).
d(2).
d(3).
```

[1] http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/ .
[2] http://www.satlib.org/solvers.html .
[3] http://www.star.dist.unige.it/∼sim/simo/ .
[4] http://www.princeton.edu/∼chaff/ .

```
b:-not a(1).
a(1):-not b.
c(X):-a(X),d(X).
```

LPARSE will produce the following grounded program

```
d(1).
d(2).
d(3).
b :- not a(1).
a(1) :- not b.
c(1) :- a(1).
c(2) :- a(2).
c(3) :- a(3).
```

After simplification step that is based on the theory described here, CMODELS will eliminate rules

```
c(2) :- a(2).
c(3) :- a(3).
```

in favor of constraints

```
:- a(2).
:- a(3).
```

If Π is a traditional program, by $At^{in}(\Pi)$ we denote the intersection of all answer sets of Π, and $At^{out}(\Pi)$ stands for the set of atoms that do not belong to any of the answer sets of Π. This section presents the theory that allows us to simplify the program when some information on sets $At^{in}(\Pi)$ and $At^{out}(\Pi)$ is available.

The following proposition shows that if some atom b occurring in the positive part of the body of some rule does not belong to any answer set then it is safe to replace this rule by the constraint $\leftarrow b$.

Proposition 1. *Any traditional program* Π *of the form*

$$a \leftarrow b,\ B$$
$$\Pi'$$

where $b \in At^{out}(\Pi)$ *has the same answer sets as*

$$\leftarrow b$$
$$\Pi'.$$

On the other hand, Proposition 2 shows that if some atom b occurring in the negative part of the body of some rule belongs to every answer set then it is safe to replace this rule by the constraint $\leftarrow not\ b$. Furthermore, if some atom b occurring in the negative part of the body of some rule does not belong to any answer set then it is safe to remove the occurrence of this atom from the body of this rule and add the constraint $\leftarrow b$ to the program.

Proposition 2. *Let* Π *be a traditional program of the form*

$$a \leftarrow not\ b,\ B$$
$$\Pi'.$$

(a) If $b \in At^{in}(\Pi)$ *then* Π *has the same answer sets as*

$$\leftarrow not\ b$$
$$\Pi'.$$

(b) If $b \in At^{out}(\Pi)$ *then* Π *has the same answer sets as*

$$\leftarrow b$$
$$a \leftarrow B$$
$$\Pi'.$$

For instance, let Π be the program

$$a \leftarrow b, not\ c$$
$$d \leftarrow not\ a$$
$$c \leftarrow not\ d.$$

The only answer set for this program is $\{d\}$. Consequently, $At^{in}(\Pi) = \{d\}$ and $At^{out}(\Pi) = \{a, b, c\}$. According to Proposition 1, Π has the same answer sets as the

following program:

$$\leftarrow b$$
$$d \leftarrow not\ a$$
$$c \leftarrow not\ d.$$

By Proposition 2 (a), Π' has the same answer sets as

$$\leftarrow b$$
$$d \leftarrow not\ a$$
$$\leftarrow not\ d.$$

This program can be further simplified using Proposition 2 (b):

$$\leftarrow b$$
$$\leftarrow a$$
$$d \leftarrow$$
$$\leftarrow not\ d.$$

In Section 8.2 we state the generalizations of Propositions 1 and 2. In Section 8.4 we present the proof of those propositions.

In Chapter 5 we described the algorithm behind the answer set solver SMODELS. The system SMODELS implements the procedures *Atleast* [Simons *et al.*, 2002, Section 7], [Simons, 2000, Section 4.1–4.3] (that can be characterized by transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Backchain False*) and *Atmost* (that can be characterized by transition rule *Unfounded*). For a traditional program Π, the procedures *Atmost* and *Atleast*, applied to Π and the empty set, find subsets of $At^{in}(\Pi)$ and $At^{out}(\Pi)$. Incorporating the *Atmost* and *Atleast* procedures in the code of CMODELS for finding these subsets of $At^{in}(\Pi)$ and $At^{out}(\Pi)$ allows us to simplify the program in accordance with Propositions 1 and 2.

The system SMODELS simplifies the program repeatedly once new information is gained about an answer set after the application of the *Atleast* and *Atmost* procedures. Within a SAT-based approach we consider the application of simplification techniques only once before the program is translated into propositional theory. Once the propositional formula is given to a SAT solver, UNIT-PROPAGATE performs simplifications on the propositional formula repeatedly.

Simplifying a program may reduce the number of program's variables and

rules, and accordingly the number of clauses created during the program's completion. From the experimental analysis that we conducted we observed that by using simplification step CMODELS usually generates at least 10% fewer atoms and around 50% fewer clauses.

6.2 Verifying Tightness

Let us note that if a program contains "trivial" rules of the form

$$a \leftarrow \ldots, a, \ldots$$

then dropping these rules from the program does not change its answer sets. We take this fact into account by eliminating such rules from a program before verifying its tightness. Obviously any program that contains a trivial rule is not tight, but dropping such rules can make a nontight program tight.

To verify whether a program is tight, CMODELS

(1) builds a subgraph of the program's dependency graph (see Section 4.5) that does not contain some "inessential" vertices:

- atoms that do not occur in the heads of program's rules, and
- atoms that do not occur in the positive parts of the bodies of the rules;

(2) uses a standard depth first search algorithm [Cormen *et al.*, 1994, Section 23.3] to detect a cycle in the subgraph.

6.3 Completion and Clausification

Recall that the completion of a program Π is defined as follows. For every rule (4.5) in Π we form the implication

$$B \rightarrow a \tag{6.1}$$

and for each atom a occurring in Π, we form the implication

$$a \rightarrow \bigvee_{B \in Bodies(\Pi, a)} B. \tag{6.2}$$

The completion $Comp(\Pi)$ of Π consists of all formulas (6.1) and (6.2).

It is clear that *CNF-Comp*(Π) — the completion $Comp(\Pi)$ converted to CNF using straightforward equivalent transformations (see Section 5.4) — can be exponentially larger than $Comp(\Pi)$. We now define an ED-transformation procedure that converts an implication (6.2) into a CNF formula by means of explicit definitions and avoids exponential growth. It is a special case of the Tseitin procedure [Tseitin, 1968] that efficiently transforms any given propositional formula to CNF form by adding new atoms corresponding to its subformulas. It does not produce a CNF equivalent to the input formula, but it gives us its conservative extension. The ED-transformation procedure adds explicit definitions for the disjunctive terms in (6.2) whenever they contain more than one atom. In other words, it introduces auxiliary atoms as abbreviations for these disjunctive terms. It then applies equivalent transformation to resulting formula and replaces these disjunctive terms by their corresponding auxiliary atoms. At last, the ED-transformation procedure converts this formula to CNF using straightforward equivalent transformations.

For instance, consider the program

$$\begin{aligned} a \leftarrow\ & b_1,\ c_1 \\ a \leftarrow\ & b_2,\ c_2 \\ a \leftarrow\ & d. \end{aligned}$$

For atom a, the implication (6.2) is

$$a \rightarrow (b_1 \wedge c_1) \vee (b_2 \wedge c_2) \vee d. \tag{6.3}$$

First, ED-transformation introduces following explicit definitions

$$\begin{aligned} aux_1 &\equiv b_1 \wedge c_1 \\ aux_2 &\equiv b_2 \wedge c_2 \end{aligned} \tag{6.4}$$

Second, ED-transformation turns implication (6.3) into the formula

$$a \rightarrow aux_1 \vee aux_2 \vee d \tag{6.5}$$

that contains two auxiliary atoms $aux_1,\ aux_2$. Third, ED-transformation clausifies

these formulas (6.4) and (6.5) as follows:

$$\begin{aligned}
&\neg a \vee aux_1 \vee aux_2 \vee d\\
&\neg aux_1 \vee b_1\\
&\neg aux_1 \vee c_1\\
&\neg b_1 \vee \neg c_1 \vee aux_1\\
&\neg aux_2 \vee b_2\\
&\neg aux_2 \vee c_2\\
&\neg b_2 \vee \neg c_2 \vee aux_2.
\end{aligned}$$

We now define for a program Π, a CNF formula *ED-Comp*(Π) that is the completion *Comp*(Π) converted to CNF using the *ED*-transformation: It consists of clauses of two kinds

1. the rules $a \leftarrow B$ of the program written as clauses

$$a \vee \overline{B},$$

2. formulas (6.2) converted to CNF using the *ED*-transformation.

The system CMODELS builds *ED-Comp*(Π) during its computation. When a satisfying assignment for *ED-Comp*(Π) is found the CMODELS algorithm eliminates all auxiliary atoms from this assignment to produce the corresponding model of the completion *Comp*(Π).

6.4 Experimental Analysis

For tight traditional programs, we implement the method discussed in this chapter in the system CMODELS. Like SMODELS, it uses the grounder LPARSE or GRINGO as its front-end. CMODELS provides an interface to four SAT solvers MINISAT, RELSAT, SIMO, and ZCHAFF. Incorporating a new SAT solver into the system does not require much programming. The fact that CMODELS may leverage on the latest developments in the SAT area is the main advantage of our SAT-based approach to computing answer sets.

In Chapter 5 we provided details on the SMODELS algorithm that is characterized by the graph SM_Π (see Section 5.2). Furthermore, in Section 5.4 we demon-

strated that SM_{Π} is almost identical to the graph $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ that characterizes an application of the basic DPLL algorithm to the program's completion that is converted to CNF by straightforward equivalent transformations (Section 5.4). The system CMODELS, on the other hand, can be best described by the graph $\text{DPL}_{\textit{ED-Comp}(\Pi)}$ (see Section 3.4). This graph differs from $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ in several ways. First, it describes an algorithm that permits conflict-driven backjumping and learning. Note that clause learning can exponentially improve the basic DPLL algorithm [Beame *et al.*, 2004]. Second, the graph considers an application of the algorithm to *ED-Comp*(Π) rather than *CNF-Comp*(Π). Gebser and Schaub [2007] demonstrated that permitting an ASP solver to make a choice on values of bodies of program rules in addition to program atoms may sometime provide an exponential overhead for the system. The introduction of auxiliary atoms that "abbreviate" bodies of program rules by means of ED-transformation allows CMODELS to perform both type of choices implicitly.

Section 6.5 provides

- the technical specifications of the system on which we conduct all experiments presented in this dissretation, and
- the details on which versions of answer set solvers were used in our experiments.

Section 6.5.1 includes a short description of the benchmarks used in the experiments. Section 6.5.2 presents the experimental results comparing the performance of the SAT-based system CMODELS with "native" answer set search algorithms implemented in SMODELS, SMODELS_{cc}, and DLV.

6.5 Systems Specifications

In this dissertation we conduct experimental analysis using a system with the following technical specifications:

- i686 GNU/Linux Ubuntu
- Two 2.99 GHz Intel Pentium D processors
- 2 GB RAM

- Limitations for the computation: 600 sec, 256 MB RAM

  In each table that reports the results

- the running time is presented in *sec*,
- *t-o*, *m-o* stand for a solver exceeding 600 sec, 256 MB respectively.

  We will report running times of

- grounder GRINGO version 2.0.3,
- answer set solver CMODELS version 3.79 using MINISAT version 2.0 beta,
- answer set solver CMODELS version 3.79 using ZCHAFF version 2007.3.12,
- answer set solver SMODELS version 2.34,
- answer set solver SMODELS$_{cc}$ version 1.08,
- grounder and answer set solver DLV version [build BEN/Oct 11 2007].

In the experiments presented in this chapter we used the grounder GRINGO to instantiate programs for the systems CMODELS, SMODELS, and SMODELS$_{cc}$. We report running time for GRINGO separately. Recall that the system DLV starts its compuation by grounding a problem. Therefore, when running time of DLV is compared with that of another solver, the running time of GRINGO should be added to the time of the solver. We also note that SMODELS$_{cc}$ can be seen as an enhanced version of SMODELS that implements learning and backjumping techniques.

In all experiments that we present in this dissertation, we consider the task of finding a single solution. Note that finding an answer set for a program that has no solutions is similar to the task of finding all solutions since a solver should traverse complete search space.

6.5.1 Benchmarks Description

ASPARAGUS[5] [Borchert *et al.*, 2004] is a benchmark platform that was created to facilitate system development in answer set programming. It aims to make a broad collection of benchmarks accessible for ASP systems. Most of the encodings and

[5] http://asparagus.cs.uni-potsdam.de/

instances used for experimental analysis in this dissertation were obtained from ASPARAGUS.

We consider several benchmarks in this chapter: *Pigeon Hole, Graph Coloring, Schur Numbers, 15-puzzle, n-queens, Blocked n-queens, Putting Numbers*. Here are short descriptions of these problems:

- In the *Pigeon Hole* problem, given p pigeons and n holes, the task is to find a unique hole for each pigeon.
- In the *Graph Coloring* problem, given a graph as a set of nodes and edges, the task is to find a way to color the nodes with n colors such that two adjacent nodes are not colored with the same color.
- A set X of integers is called sum-free if for any elements x and y from X, $x + y$ is not in X. The largest integer n such that the set $\{1, 2, ..., n\}$ can be partitioned into p sum-free sets is called the Schur number $S(p)$. The *Schur Numbers* problem is to decide whether an integer n satisfies $n \leq S(p)$.
- In the *15-puzzle* problem, a 4 x 4 grid is filled in some way with 15 movable tiles labeled $1, \ldots, 15$. One square is left blank. Tiles can be rearranged by moving a tile into a blank square, assuming the two locations share an edge. The goal is to rearrange the tiles so that they are arranged from 1 to 15 in the row-major order, with the blank square occupying the top left corner.
- In the *n-queens* problem, we have an n x n board and n queens. Each square on the board can hold at most one queen. A conflict arises when any two queens are assigned to the same row, column, or diagonal. In the *n-queens* the task is to assign the n queens to the squares of the board in a conflict-free manner.
- The *Blocked n-queens* problem is a variant of the n-queens problem. where some squares on the board are blocked and cannot hold any queen. A solution to the Blocked n-queens problem is an assignment of the n queens to the non-blocked squares of the board in a conflict-free manner.
- In the *Putting Numbers* problem, the input is a set of m-digit numbers and the task is to construct a n x n square where all of the numbers occur. The numbers may occur horizontally or vertically and in both directions.

6.5.2 Benchmarks Results

All benchmark instances that we consider for experimental analysis in this chapter are encoded as tight traditional programs.

Figure 6.1 reports the running times for GRINGO, CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, SMODELS$_{cc}$, and DLV on Pigeon Hole, Graph Coloring, Schur Numbers, and 15-Puzzle problems. In all presented experiments, we run all systems on identical encodings of the problems. Figure 6.2 reports the running times for GRINGO, CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, SMODELS$_{cc}$ on n-queens, Blocked n-queens, and Putting Numbers. We do not present running times for DLV in the latter figure because the encodings of n-queens, Blocked n-queens, and Putting Numbers for the grounder GRINGO used in experiments contain constructs not allowed by the input language of DLV. Instance names may be decoded as follows:

- *pigeon.p9h8* suggests that the considered instance is an encoding of Pigeon Hole problem with 9 pigeons and 8 holes,
- *color.n100.3* suggests that the considered instance is an encoding of Graph Coloring problem with 100 nodes and 3 colors,
- *schur.p4n43* suggests that the considered instance is an encoding of Schur Number problem where p is 4 and n is 43,
- *15-puzzle.1* suggests that the considered instance is an encoding of 15-puzzle problem,
- *queens.18* suggests that the considered instance is an encoding of n-queens problem where n is 18,
- *bqueens.50.1642408561* suggests that the considered instance is an encoding of Blocked n-queens problem where n is 50 and *1642408561* corresponds to an *id* assigned by ASPARAGUS to this instance,
- *pn.gsquare-3-10-2-8* suggests that the considered instance is an encoding of Putting Numbers problem and *gsquare-3-10-2-8* corresponds to an *id* assigned by ASPARAGUS to this instance.

Instance	GRINGO	CMODELS		SMODELS	SMODELS$_{cc}$	DLV
		MINISAT	ZCHAFF			
pigeon.p9h8	0.0	0.56	0.45	4.8	310.21	15.48
pigeon.p10h9	0.01	6.28	3.68	47.19	t-o	161.53
pigeon.p11h10	0.01	85.02	12.29	509.27	t-o	t-o
pigeon.p12h11	0.02	t-o	42.06	t-o	t-o	t-o
color.n100.3	0.03	0.03	0.02	0.02	0.03	0.02
color.n300.3	0.14	0.1	0.08	0.07	0.12	0.08
color.n600.3	0.41	0.24	0.19	0.16	0.27	0.19
color.n1000.3	0.99	0.43	0.34	0.3	0.48	0.31
color.n3000.3	7.56	1.48	1.16	0.97	1.55	1.18
color.n6000.3	33.63	3.15	2.42	2.0	3.12	2.44
color.n100.4	0.04	0.05	0.03	0.12	0.42	236.59
color.n300.4	0.19	0.16	0.13	1.47	4.53	t-o
color.n600.4	0.53	0.39	0.35	6.72	19.88	5.86
color.n1000.4	1.27	0.74	0.66	21.47	60.47	17.7
color.n3000.4	9.91	8.94	198.01	211.58	577.43	154.53
color.n6000.4	44.81	t-o	t-o	t-o	t-o	t-o
schur.p4n43	0.06	0.15	0.09	0.58	0.79	t-o
schur.p4n44	0.06	0.31	0.64	32.58	47.07	t-o
schur.p4n45	0.06	0.27	0.67	44.3	48.99	t-o
schur.p5n100	0.34	4.51	3.57	t-o	t-o	t-o
schur.p5n110	0.41	16.91	11.35	t-o	t-o	t-o
schur.p5n120	0.48	362.79	595.56	t-o	t-o	t-o
schur.p5n130	0.57	t-o	t-o	t-o	t-o	t-o
15-puzzle.1	0.38	9.76	13.94	t-o	t-o	t-o
15-puzzle.2	0.38	45.55	t-o	t-o	t-o	t-o
15-puzzle.3	0.38	5.04	412.65	t-o	t-o	t-o
15-puzzle.4	0.37	93.93	t-o	t-o	t-o	t-o
15-puzzle.5	0.38	112.25	t-o	t-o	t-o	t-o
15-puzzle.6	0.39	127.45	565.07	t-o	t-o	t-o
15-puzzle.7	0.38	59.73	t-o	t-o	t-o	t-o
15-puzzle.8	0.39	3.46	t-o	374.39	t-o	t-o
15-puzzle.9	0.41	7.85	t-o	t-o	t-o	t-o
15-puzzle.10	0.43	118.16	t-o	t-o	t-o	t-o
15-puzzle.11	0.43	206.34	t-o	t-o	t-o	t-o
15-puzzle.12	0.43	1.91	115.12	t-o	t-o	t-o

Figure 6.1: Pigeon Hole, Graph Coloring, Schur Numbers, 15 Puzzle; runtimes of GRINGO, CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, SMODELS$_{cc}$, and DLV.

Instance	GRINGO	CMODELS		SMODELS	SMODELS$_{cc}$
		MINISAT	ZCHAFF		
queens.18	0.16	0.22	0.15	6.8	126.98
queens.22	0.3	0.49	0.3	t-o	t-o
queens.24	0.4	0.71	0.39	t-o	t-o
queens.28	0.68	1.26	0.67	t-o	t-o
queens.32	1.08	2.19	0.95	t-o	t-o
queens.36	1.64	3.41	1.41	t-o	t-o
bqueens.50.1642398261	5.45	157.11	t-o	t-o	t-o
bqueens.50.1642402587	5.56	87.1	t-o	t-o	t-o
bqueens.50.1642406388	5.44	104.12	t-o	t-o	t-o
bqueens.50.1642406727	5.2	266.84	t-o	t-o	t-o
bqueens.50.1642407126	5.37	485.78	t-o	t-o	t-o
bqueens.50.1642407857	5.47	119.3	t-o	t-o	t-o
bqueens.50.1642408561	5.4	201.16	t-o	t-o	t-o
pn.gsquare-3-10-2-8	0.02	0.02	0.02	0.06	0.28
pn.gsquare-4-11-3-8	0.04	0.07	0.71	112.92	260.81
pn.gsquare-4-12-3-8	0.04	0.07	6.78	6.18	296.18
pn.gsquare-4-14-3-8	0.05	0.07	0.07	10.61	127.4
pn.gsquare-4-19-3-8	0.07	0.98	2.09	223.02	t-o
pn.gsquare-4-22-3-8	0.09	2.48	4.6	105.38	t-o
pn.gsquare-4-24-3-8	0.1	0.3	3.81	402.48	t-o
pn.gsquare-4-9-3-8	0.03	0.04	0.03	0.54	0.92
pn.gsquare-5-12-4-8	0.07	82.4	213.96	t-o	t-o
pn.gsquare-7-25-6-8	0.31	t-o	206.71	t-o	t-o

Figure 6.2: n-queens, Blocked n-queens, Putting Numbers; runtimes of GRINGO, CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, SMODELS$_{cc}$.

From the tables in Figures 6.1 and 6.2 we can make two conclusions. First, CMODELS often outperforms "native" answer set search algorithms implemented in SMODELS, SMODELS$_{cc}$, and DLV. Second, there is often a considerable difference between the performance of CMODELS with MINISAT and with ZCHAFF. Which one of the two SAT solvers is more efficient depends on the particular problem instance to be solved. For example, CMODELS using ZCHAFF finds solutions to all instances of Putting Numbers problem while CMODELS using MINISAT is not able to find the solution for one of the instances within the time limit. On the other hand, on all considered instances of the Blocked n-queens problems CMODELS using ZCHAFF is

not able to find a solution within the time limit while CMODELS using MINISAT computes solutions to all instances within permitted time. This demonstrates the main plus of the SAT-based answer set solving approach: an answer set system like CMODELS may take advantage by using a SAT solver that is best suited for a specific problem.

Chapter 7

Background: Choice and Weight Rules

Good answer set programming systems, such as SMODELS or DLV, support not only traditional rules but also some useful "additional" constructs. For instance, Niemelä and Simons [2000] introduced choice rules and weight constraints for the system SMODELS. The answer set solver DLV features weak constraints [Buccafurri *et al.*, 1997] and aggregate predicates [Dell'Armi *et al.*, 2003]. The applicability of answer set programming has been widened once the basic language of traditional programs was extended to more expressive constructs. For instance, choice rules of SMODELS allow us to represent choices and restrictions on solutions in a concise manner.

As mentioned before, the grounder LPARSE and the ASP solver SMODELS interpret such constructs as choice and weight constraint rules. In this chapter we define their semantics in terms of rules with nested expressions, introduced in [Lifschitz *et al.*, 1999]. This definition is different from the one given originally by Niemelä and Simons but is equivalent to it, as demonstrated in [Ferraris and Lifschitz, 2005].

Section 7.1 introduces programs with nested expressions and extends the answer set semantics to such programs. Sections 7.2 and 7.3 describe the semantics of choice, weight and cardinality constraint rules in terms of programs with nested expressions. Section 7.4 introduces the syntax of choice and weight rules that are allowed in the input language of LPARSE. Section 7.5 introduces semi-traditional rules – a special class of rules with nested expressions. Choice and weight rules can

be both understood as abbreviations for semi-traditional rules. At last, Section 7.6 extends the notions of completion and tightness to programs with semi-traditional rules.

Although we use rules with nested expressions to define the semantics of programs with choice and weight constraint rules, arbitrary programs with nested expressions are more general. For instance, a rule with nested expressions can contain a disjunction in its head. In order to define the semantics of a program with choice or weight rules, a disjunction in the head of a rule is not needed. Further in the dissertation we will investigate the applicability of the SAT-based method to programs with nested expressions that also allow a disjunction in their head.

The next chapter demonstrates how SAT solvers can be used to process programs with choice and weight constraint rules.

7.1 Programs with Nested Expressions

Elementary expressions are atoms and the symbols $\bot$ (*False*) and $\top$ (*True*). *Nested expressions* are built from elementary expressions using the unary connective *not* (negation as failure) and the binary connectives conjunction (recall that we identify comma with conjunction) and disjunction.

A program with nested expressions, also called *nested*, consists of rules of the form

$$Head \leftarrow Body \tag{7.1}$$

where both $Body$ and $Head$ are nested expressions. We call such rules *nested* or *rules with nested expressions*.

The definition of satisfaction for a rule, a head, or a body of a rule is the usual definition of satisfaction in propositional logic, with "*not*" understood as negation, and $Head \leftarrow Body$ as the material implication $Body \rightarrow Head$. We say that a set X of atoms *satisfies* a program Π (symbolically, $X \models \Pi$) if X satisfies every rule (7.1) in Π. We say also that such set X is a *model* of Π.

Consider a set X of atoms and a formula F. The *reduct* F^X of F with respect to X is defined as follows:

- for elementary F, $F^X = F$
- $(F \wedge G)^X = F^X \wedge G^X$

- $(F \vee G)^X = F^X \vee G^X$
- $(not\ F)^X = \begin{cases} \top, & \text{if } X \models F, \\ \bot, & \text{otherwise} \end{cases}$

The *reduct* Π^X of a program Π with respect to a set X of atoms is the set of rules

$$Head^X \leftarrow Body^X$$

for all rules (7.1) in Π.

Note that if Π is a traditional program then this definition of the reduct is equivalent to the definition given in Section 4.2 because the occurrence of $\bot$ in the body of a rule is equivalent to removing the rule and the occurrence of $\top$ in the body of a rule is equivalent to removing $\top$ from the body.

A set X of atoms is an *answer set*, also called *stable model* of Π, if X is a minimal set of atoms satisfying the reduct Π^X.

As in case of traditional programs, an answer set of a program with nested expressions is also always a model of this program.

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow not\ not\ a \\ b \leftarrow not\ a. \end{array} \tag{7.2}$$

Consider the set $\{a\}$. The reduct $\Pi^{\{a\}}$ is

$$\begin{array}{l} a \leftarrow \top \\ b \leftarrow \bot. \end{array} \tag{7.3}$$

Set $\{a\}$ satisfies the reduct and is minimal among the sets satisfying the reduct. Hence, $\{a\}$ is an answer set of Π. Consider now the set $\{a, b\}$. The reduct $\Pi^{\{a,b\}}$ is identical to (7.3). Nevertheless, $\{a, b\}$ does not satisfy the reduct and hence is not an answer set of Π. In fact, this program has only one answer set besides $\{a\}$ – set $\{b\}$.

Note that unlike in classical logic where double negation is inessential and can be removed from an expression, *not not* plays an important role in programs with nested expressions. Consider the traditional program obtained from (7.2) by

dropping the double negation

$$\begin{aligned} a &\leftarrow a \\ b &\leftarrow not\ a. \end{aligned} \tag{7.4}$$

The reduct of this program on $\{a\}$ is

$$\begin{aligned} a &\leftarrow a \\ b &\leftarrow \bot \end{aligned}$$

so that set $\{a\}$ is not an answer set of program (7.4).

7.2 Choice Rules

Choice rules [Syrjanen, 2003, Sections 5.4] have the form

$$\{a_0, \ldots, a_k\} \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m \tag{7.5}$$

where each a_i and b_i is an atom. Intuitively, this rule expresses that if its body is satisfied then any subset of $\{a_0, \ldots, a_k\}$ *may* belong to an answer set. Originally the semantics of choice rules was defined in [Niemelä and Simons, 2000]. We follow the alternative definition of the semantics of choice rules, proposed by Ferraris and Lifschitz [2005]. They treat the choice rule (7.5) as an abbreviation for the following rule with nested expressions:

$$(a_0 \vee not\ a_0), \ldots, (a_k \vee not\ a_k) \ \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m \tag{7.6}$$

They showed that this definition is equivalent to the semantics in [Niemelä and Simons, 2000].

For instance, the choice rule

$$\{a, b\}$$

is shorthand for

$$(a \vee\ not\ a),\ (b \vee\ not\ b) \leftarrow .$$

One-rule program has four answer sets: $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$.

7.3 Weight and Cardinality Constraint Rules

Weight (constraint) rules have the form

$$a \leftarrow L\,\{F_1 = w_1, \ldots, F_n = w_n\} \tag{7.7}$$

where a is an atom or the symbol $\bot$, each F_i is an atom b_i or the formula *not* b_i, and L (lower bound) and $w_1, \ldots, w_n$ (weights) are integers. This is a special case of weight rules in the sense of [Syrjanen, 2003, Section 5.4]. Intuitively rule (7.7) states that a must hold whenever the sum of the "eligible" weights is greater than or equal to lower bound L, where by eligible weights we understand these w_i for which F_i is satisfied. Weight rules with all weights $w_1, \ldots, w_n$ equal to 1 are called *cardinality constraints* [Syrjanen, 2003, Section 5.4]. The semantics of weight rules was originally introduced by Niemelä and Simons [2000]. In this dissertation we adopt the view of Ferraris and Lifschitz [2005] that a weight rule is understood as an abbreviation for a rule with nested expressions, as follows.

Consider the expression

$$< F_1, \ldots, F_n >: X \tag{7.8}$$

where X is set of subsets of $\{1, \ldots, n\}$, each F_i is an atom or the formula *not* b_i (where b_i is an atom). Expression (7.8) is an abbreviation for the formula

$$\bigvee_{I \in X} \Big(\bigwedge_{i \in I} F_i \Big) \tag{7.9}$$

Ferraris and Lifschitz [2005] treat weight rule (7.7) as an abbreviation for the rule with nested expressions

$$a \leftarrow < F_1, \ldots, F_n >: \{I : L \leq \textstyle\sum_{i \in I} w_i\} \tag{7.10}$$

where I ranges over subsets of $\{1, \ldots, n\}$.

For instance, we understand the weight rule

$$a \leftarrow 3\,\{b = 3, c = 2, d = 2\} \tag{7.11}$$

as an abbreviation for the rule with nested expressions

$$a \leftarrow < b, c, d >: \{\{1\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

In other words, the weight rule (7.11) stands for the following rule with nested expressions:

$$a \leftarrow b \vee (b,c) \vee (b,d) \vee (c,d) \vee (b,c,d).$$

7.4 The Input Language of Lparse

The grounder LPARSE accepts rules of even more general form than the choice and weight rules discussed above. For instance, following rules are also allowed:

$$a \leftarrow L\,\{F_1 = w_1, \ldots, F_n = w_n\}\,U \tag{7.12}$$

$$L\,\{a_1, \ldots, a_n\}\,U \leftarrow Body. \tag{7.13}$$

Rule (7.12) is similar to weight rule (7.7) except that upper bound U (an integer) is specified on the right hand side of the weight expression. Intuitively rule (7.12) states that a (an atom or symbol $\perp$) must hold whenever the sum of the eligible weights is between the lower bound L and upper bound U.

Rule (7.13), on the other hand, is similar to (7.5), except that additional restrictions are specified by the lower bound L and upper bound U (L and U are integers). The rule expresses that any subset of atoms $\{a_0, \ldots, a_k\}$ with at least L and at most U elements may belong to an answer set when the $Body$ of a rule is satisfied.

Rules (7.12) and (7.13) are eliminated by LPARSE in favor of choice and weight constraint rules of type (7.5) and (7.7) (see [Syrjanen, 2003]). Therefore within this dissertation we only consider the case of choice rules (7.5) and weight constraint rules (7.7).

For instance, consider a weight rule

$$a \leftarrow 3\{b = 3, c = 2, d = 2\}4.$$

Given a choice rule $\{b, c, d\}$, LPARSE produces the following set of choice and weight rules:

```
a :- __int0, not __int1.
__int0 :- 3 [ b=3, c=2, d=2 ].
__int1 :- 5 [ b=3, c=2, d=2 ].
{ d, c, b }.
```

7.5 Semi-Traditional Programs

We say that a rule with nested expressions is *semi-traditional* if it has the following form

$$a \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n, \tag{7.14}$$

where a is an atom or symbol $\bot$ and each b_i is an atom. Note that if $n = m$ then a semi-traditional rule is identical to a traditional rule. A program is *semi-traditional* if it consists of semi-traditional rules.

The reason why this class of programs is important can be described using the concept of strong equivalence for logic programs, introduced by Lifschitz et al [2001]. Two logic programs Π_1 and Π_2 are *strongly equivalent* if for every logic program Π, programs $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ have the same answer sets. We use this terminology to state the results in this section.

It turns out that both choice and weight rules can be rewritten as strongly equivalent semi-traditional programs. In fact, any weight rule can be rewritten as a set of traditional rules; semi-traditional rules (7.14) with $n > m$ are needed for choice rules only.

For instance, recall that the weight rule

$$a \leftarrow 3\,\{b = 3, c = 2, d = 2\}$$

stands for the following rule with nested expressions

$$a \leftarrow b \vee (b, c) \vee (b, d) \vee (c, d) \vee (b, c, d).$$

This rule is strongly equivalent to the set of traditional rules

$$\begin{aligned} a &\leftarrow b \\ a &\leftarrow b, c \\ a &\leftarrow b, d \\ a &\leftarrow c, d \\ a &\leftarrow b, c, d. \end{aligned}$$

On the other hand, consider choice rule (7.5)

$$\{a_0, \ldots, a_k\} \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m.$$

It abbreviates the rule with nested expressions

$$(a_0 \vee not\ a_0)\ , \ldots,\ (a_k \vee not\ a_k)\ \leftarrow b_1, \ldots, b_l,\ not\ b_{l+1}, \ldots,\ not\ b_m.$$

This rule is strongly equivalent to the semi-traditional program:

$$\begin{aligned} a_0\ &\leftarrow b_1, \ldots, b_l,\ not\ b_{l+1}, \ldots,\ not\ b_m,\ not\ not\ a_0 \\ &\ldots \\ a_k\ &\leftarrow b_1, \ldots, b_l,\ not\ b_{l+1}, \ldots,\ not\ b_m,\ not\ not\ a_k. \end{aligned}$$

7.6 Tightness and Completion for Semi-Traditional Programs

In this section we review the definitions of completion and tightness for semi-traditional programs. This is sufficient for our purposes, because the CMODELS algorithm eliminates choice and weight rules in favor of semi-traditional rules as described in the previous section.

Recall that a semi-traditional rule (7.14) has the form

$$a \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n.$$

We will write this rule in two other forms

$$a \leftarrow B, \tag{7.15}$$

$$a \leftarrow D, F, \tag{7.16}$$

where, as in case of traditional programs, the positive part D of the body is

$$b_1, \ldots, b_l$$

and the negative part F is the list of the remaining terms of B

$$not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n.$$

We identify the body of a rule (7.14) with the conjunction

$$b_1 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \ldots \neg b_m \wedge \neg\neg b_{m+1} \wedge \cdots \wedge \neg\neg b_n.$$

Note that this conjunction is equivalent in classical logic to

$$b_1 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \ldots \neg b_m \wedge b_{m+1} \wedge \cdots \wedge b_n.$$

The definitions of completion and supported model for a semi-traditional program is the same as in Section 4.4. This definition of completion is a special case of a definition given by Lloyd and Topor [1984].

As we have seen before removing *not not* changes the answer sets of a program (see example in the end of Section 7.1), but it does not change the models of completion and hence the supported models of a program. For instance, consider program (7.2) from Section 7.1. Its completion is

$$\begin{aligned} &\neg\neg a \rightarrow a \\ &\neg a \rightarrow b \\ &a \rightarrow \neg\neg a \\ &b \rightarrow \neg a. \end{aligned} \tag{7.17}$$

Two of these implications are, of course, tautologies. The completion has two models: $\{a\}$, $\{b\}$. The completion of program (7.4) obtained from (7.2) by dropping the double negation is equivalent to (7.17).

The definitions of the dependency graph and tightness given in Section 4.5 for traditional programs can be extended to semi-traditional programs, if we refer

to a program Π as a semi-traditional program and to a rule in the program as a semi-traditional rule of the form (7.16).

For instance, consider program (7.2). Its dependency graph contains no edges. The graph is trivially acyclic and hence the program is tight. On the other hand, program (7.4) contains one cycle from node a to itself. Hence the program is not tight. This example illustrates that removing *not not* may also change the tightness property of a program similarly as it may change the answer sets of a program.

Theorem 9 (Theorem on Tight Semi-Traditional Programs)**.** *For any tight semi-traditional program* Π *and any set* X *of atoms,* X *is an answer set for* Π *if and only if* X *satisfies* $Comp(\Pi)$*.*

Theorem 9 is a generalization of Theorem 4 in Section 4.5 and a special case of a theorem due to Erdem and Lifschitz [2001].

For instance, program (7.2) is tight. From Theorem 9, it follows that the two models of the program's completion $\{a\}$, $\{b\}$ are also its answer sets. On the other hand, program (7.4) is not tight. It has only one answer set, $\{b\}$.

Chapter 8

Extending Cmodels Algorithm to Choice and Weight Rules

This chapter generalizes the findings of Chapter 6 on using SAT for traditional tight programs to the case of programs with extended syntax. First, Section 8.1 provides the details on how the CMODELS algorithm eliminates weight rules in favor of traditional rules using auxiliary atoms. Section 8.2 extends the findings on program simplification from Section 6.1 to the case of programs with choice and weight rules and Section 8.4 presents the proofs for these results. Section 8.3 shows how SAT can be applied to tight programs of a more general kind. Finally Section 8.5 compares the performance of CMODELS and SMODELS on such programs.

8.1 Translating Weight Rules

In Section 7.5 we showed how a weight rule can be rewritten as a set of traditional rules. That translation is, however, not efficient because its result can be exponentially large. For instance, according to the method illustrated in Section 7.5 a cardinality constraint rule

$$a \leftarrow 0\{b_1, b_2, \ldots, b_n\}n$$

can be replaced by 2^n traditional rules. Therefore, in the CMODELS algorithm we used another translation, proposed by Ferraris and Lifschitz [2005]. Their translation $[\Pi]$ of a program Π uses auxiliary atoms to eliminate the weight rules from Π

in favor of traditional rules. This is similar to the idea behind the Tseitin procedure [Tseitin, 1968] discussed in Section 6.3. Strictly speaking, [Π] is not equivalent to Π, but it has a conservative extension property: dropping the newly introduced atoms from the answer sets of [Π] provides the answer sets of Π. The complete description of the transformation is not reproduced here (see [Ferraris and Lifschitz, 2005]), but in this section we give an example and discuss some implementation details.

For instance, the rule

$$a \leftarrow 3\,\{b = 3, c = 2, d = 2\}$$

is first replaced by four simpler rules:

$$\begin{aligned}
&a \leftarrow aux1\\
&aux1 \leftarrow 3\,\{c = 2, d = 2\}\\
&a \leftarrow b, aux2\\
&aux2 \leftarrow 0\,\{c = 2, d = 2\}.
\end{aligned}$$

Intuitively, the first pair of rules corresponds to the answer sets that don't contain b, and the second pair to the answer sets that do. The auxiliary variable $aux1$ is an "abbreviation" for $3\,\{c = 2, d = 2\}$, and $aux2$ stands for $0\,\{c = 2, d = 2\}$. Since the last expression is identically True, $aux2$ can be dropped from the program:

$$\begin{aligned}
&a \leftarrow aux1\\
&aux1 \leftarrow 3\,\{c = 2, d = 2\}\\
&a \leftarrow b.
\end{aligned}$$

The next step is similar:

$$\begin{aligned}
&a \leftarrow aux1\\
&aux1 \leftarrow aux3\\
&aux3 \leftarrow 3\,\{d = 2\}\\
&aux1 \leftarrow c, aux4\\
&aux4 \leftarrow 1\,\{d = 2\}\\
&a \leftarrow b.
\end{aligned}$$

Here $aux3$ stands for $3\,\{d = 2\}$, and $aux4$ stands for $1\,\{d = 2\}$. Since $3\,\{d = 2\}$ is identically *False*, the program can be simplified as follows:

$$
\begin{aligned}
&a \leftarrow aux1\\
&aux1 \leftarrow c, aux4\\
&aux4 \leftarrow 1\,\{d = 2\}\\
&a \leftarrow b.
\end{aligned}
$$

Finally, we use the fact that $1\{d = 2\}$ can be equivalently replaced by d, and arrive at a set of basic rules:

$$
\begin{aligned}
&a \leftarrow aux1\\
&aux1 \leftarrow c, aux4\\
&aux4 \leftarrow d\\
&a \leftarrow b.
\end{aligned}
$$

Every auxiliary atom introduced in the process of translating weight constraint rule (7.7) into a set of traditional rules "abbreviates" an expression that differs from the body

$$L\,\{F_1 = w_1, \ldots, F_m = w_m\}$$

of that rule in two ways: the lower bound L is replaced by an integer between 1 and L, and the list of equalities in the brackets is replaced by its proper non-empty suffix. It follows that the number of auxiliary atoms in the translation of rule (7.7) does not exceed $L(n-1)$. If (7.7) is a cardinality constraint rule ($w_1 = \cdots = w_n = 1$) then we can also say that the lower bound is replaced by an integer between $L-n+1$ and L, so that the number of auxiliary atoms in the translation of a cardinality constraint rule has the upper bound $n(n-1)$.

8.2 Simplifying Programs with Nested Expressions

As in Section 6.1, given a nested program Π, by $At^{in}(\Pi)$ we denote the intersection of all answer sets of Π, and $At^{out}(\Pi)$ stands for the set of atoms that do not belong to any of the answer sets of Π. Recall that we view weight rules as abbreviations for the programs with nested expressions. The propositions below generalize Propositions 1 and 2 to programs with nested expressions.

Proposition 1 (general form). *Let* Π *be a nested program of the form*

$$\begin{array}{l} H \leftarrow L\,\{b_0 = w_0,\ F_1 = w_1, \ldots,\ F_n = w_n\} \\ \Pi' \end{array}$$

where H *is an atom, symbol* $\bot$*, or an expression of the form*

$$(a_0 \vee not\ a_0)\ , \ldots,\ (a_k \vee not\ a_k)\ ,$$

each F_i *is an atom* b_i *or formula not* b_i*. If* $b_0 \in At^{out}(\Pi)$ *then* Π *has the same answer sets as the program*

$$\begin{array}{l} \leftarrow b_0 \\ H \leftarrow L\,\{F_1 = w_1, \ldots, F_n = w_n\} \\ \Pi'. \end{array}$$

In the special case when H is an atom or $\bot$, all the weights $w_i = 1$, and $L = n + 1$, the rule

$$H \leftarrow L\,\{b_0 = w_0,\ F_1 = w_1, \ldots,\ F_n = w_n\}$$

can be rewritten as the traditional rule

$$H \leftarrow b_0,\ F_1, \ldots,\ F_n.$$

Therefore, Proposition 1 (general form) can be seen as a generalization of Proposition 1 (Section 6.1). Similarly, Proposition 2 (general form) below is a generalization of Proposition 2 (Section 6.1).

Also note that in the case when H has the form

$$(a_0 \vee not\ a_0)\ , \ldots,\ (a_k \vee not\ a_k)\ ,$$

all the weights $w_i = 1$, and $L = n + 1$, the rule

$$H \leftarrow L\,\{b_0 = w_0,\ F_1 = w_1, \ldots,\ F_n = w_n\}$$

can be rewritten as the choice rule

$$H \leftarrow b_0,\ F_1, \ldots,\ F_n.$$

Proposition 2 (general form). *Let* Π *be a nested program of the form*

$$\begin{array}{l} H \leftarrow L\,\{not\ b_0 = w_0,\ F_1 = w_1, \ldots,\ F_n = w_n\} \\ \Pi' \end{array}$$

where H *is an atom, symbol* $\bot$*, or an expression of the form*

$$(a_0 \vee not\ a_0)\ , \ldots,\ (a_k \vee not\ a_k)\ ,$$

each F_i *is an atom or formula not* b_i *(*b_i *is an atom).*
(a) If $b_0 \in At^{in}(\Pi)$ *then* Π *has the same answer sets as*

$$\begin{array}{l} \leftarrow not\ b_0 \\ H \leftarrow L\,\{\ F_1 = w_1, \ldots,\ F_n = w_n\} \\ \Pi'. \end{array}$$

(b) If $b_0 \in At^{out}(\Pi)$ *then* Π *has the same answer sets as*

$$\begin{array}{l} \leftarrow b_0 \\ H \leftarrow L - w_0\,\{\ F_1 = w_1, \ldots,\ F_n = w_n\} \\ \Pi'. \end{array}$$

For instance, let Π be the program

$$\begin{array}{l} \{a\} \\ c \leftarrow 4\{a = 1, b = 1, not\ d = 2\}. \end{array}$$

The only two answer sets for this program are $\{a\}$ and $\emptyset$. Consequently, $At^{in}(\Pi) = \emptyset$ and $At^{out}(\Pi) = \{b, c, d\}$. According to Proposition 1 (general form), Π has the same

answer sets as a program

$$\{a\}$$
$$\leftarrow b$$
$$c \leftarrow 4\{a = 1, not\ d = 2\}.$$

By Proposition 2 (general form) (b), Π' has the same answer sets as

$$\{a\}$$
$$\leftarrow b$$
$$\leftarrow d$$
$$c \leftarrow 2\{a = 1\}.$$

If we know subsets of $At^{in}(\Pi)$ and $At^{out}(\Pi)$ then Proposition 1 (general form) and Proposition 2 (general form) allow us to simplify Π.

8.3 Cmodels Algorithm for Programs with Choice and Weight Rules

Consider a program Π that consists of the weight constraints and choice rules. After eliminating weight constraints in favor of traditional rules by means of auxiliary atoms as described in Section 8.1, and transforming the choice rules into sets of semi-traditional rules as in Section 7.5, Π becomes semi-traditional. If the result of this transformation is tight then, in view of Theorem 9, the answer sets of Π can be found by a SAT solver.

We have enhanced CMODELS so that it can handle programs with choice and weight rules. The CMODELS algorithm performs the following steps during its execution:

1. simplifies the program,
2. eliminates the choice rules in favor of semi-traditional rules,
3. eliminates the weight rules in favor of traditional rules using auxiliary atoms,
4. verifies the tightness of the program,

5. produces its completion,

6. clausifies the completion,

7. invokes a SAT solver, and

8. interprets its output.

This algorithm extends the CMODELS algorithm for traditional programs, presented in Section 6, by steps 2 and 3.

The simplification step of the CMODELS algorithm uses the SMODELS procedures *Atmost* and *Atleast* [Simons *et al.*, 2002, Section 7], [Simons, 2000, Section 4.1–4.3], Proposition 1 (general form) and Proposition 2 (general form). The tightness verification, completion, and clausification steps are performed as described in Sections 6.2 and 6.3. Tightness verification is performed after an additional strongly equivivalent transformation on a program. This additional step may possibly transfrom a nontight program into a tight one. We describe this procedure below.

The output of LPARSE often contains choice rules of the form

$$\{a\}$$

where a is an atom. Recall that this rule is strongly equivalent (Section 7.5) to the semi-traditional rule

$$a \leftarrow not\ not\ a. \tag{8.1}$$

If a program contains rule (8.1) then replacing all occurrences of a in the positive parts of the bodies of program's rules by *not not a* does not change program's answer sets:

Proposition 3. *Any nested program of the form*

$$H \leftarrow a,\ B$$
$$a \leftarrow not\ not\ a$$

is strongly equivalent to the program

$$H \leftarrow not\ not\ a,\ B$$
$$a \leftarrow not\ not\ a.$$

Proposition 3 allows us, in some cases, to transform a nontight program into tight. For instance, in application to the nontight program

$$a \leftarrow not\ not\ a$$
$$a \leftarrow b$$
$$b \leftarrow a$$

this transformation inserts *not not* in front of a in its last rule, and the program becomes tight.

8.4 Proofs of Proposition 1 (general form), Proposition 2 (general form), and Proposition 3

As discussed in Section 7.5 two logic programs Π_1 and Π_2 are *strongly equivalent* if for every logic program Π, programs $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ have the same answer sets. In [Lifschitz *et al.*, 2001], the authors demonstrated that the verification of strong equivalence can be accomplished by checking the equivalence of formulas representing logic programs in a monotonic logic, called the logic of here-and-there. This logic is intermediate between classical logic and intuitionistic logic. Review of the logic of here-and-there and intuitionistic logic is out of the scope of this presentation. The interested reader will find the introduction to the logic of here-and-there in [Lifschitz *et al.*, 2001] and the introduction to intuitionistic logic in [Moschovakis, 2008]. We can think of nested rules (7.1) discussed in Section 7.1 as propositional formuals: replace every *not* with $\neg$, every comma with $\wedge$, and trun every rule

$$Head \leftarrow Body$$

into the implication

$$Body \rightarrow Head.$$

In this view any nested program can be identified with the set of propositional formulas.

Theorem 10 (Theorem 1 in [Lifschitz *et al.*, 2001])**.** *For any nested programs* Π_1 *and* Π_2*,* Π_1 *is strongly equivalent to* Π_2 *if and only if* Π_1 *is equivalent to* Π_2 *in the logic of here-and-there.*

The assertion of the Proposition 3 immediately follows from Theorem 10 due to the fact that the two programs are intuitionistically equivalent.

The following theorem in [Lifschitz *et al.*, 1999] tells us how adding a constraint to a program affects the collection of its answer sets.

Theorem on Constraints. *For any nested program* Π *and nested expression* F*, a set* X *of atoms is an answer set for* $\Pi \cup \{\leftarrow F\}$ *if and only if* X *is an answer set for* Π *and does not satisfy* F*.*

Proposition 1 (general form). *Let* Π *be a nested program of the form*

$$\begin{array}{l} H \leftarrow L\,\{b_0 = w_0,\ F_1 = w_1, \ldots,\ F_n = w_n\} \\ \Pi' \end{array}$$

where H *is an atom, symbol* $\bot$*, or an expression of the form*

$$(a_0 \vee not\ a_0)\ , \ldots,\ (a_k \vee not\ a_k)\ ,$$

each F_i *is an atom* b_i *or formula not* b_i*. If* $b_0 \in At^{out}(\Pi)$ *then* Π *has the same answer sets as the program*

$$\begin{array}{l} \leftarrow b_0 \\ H \leftarrow L\,\{F_1 = w_1, \ldots, F_n = w_n\} \\ \Pi'. \end{array} \tag{8.2}$$

Proof. From Theorem on Constraints and the fact that $b_0 \in At^{out}(\Pi)$ it follows that Π has the same answer sets as $\Pi \cup \{\leftarrow b_0\}$. On the other hand, $\Pi \cup \{\leftarrow b_0\}$ is equivivalent to program (8.2) in the logic of here-and-there. By Theorem 10, $\Pi \cup \{\leftarrow b_0\}$ is strongly equivivalent to program (8.2). Consequently, Π has the same answer sets as program (8.2). □

The proof of Proposition 2 (general form) follows the lines of the proof above.

8.5 Experimental Analysis

As in the case of traditional programs we conduct experimental analysis using the system whose technical specifications are presented in Section 6.5. We compare the performance of CMODELS using MINISAT and CMODELS using ZCHAFF with that of SMODELS. Details on the versions of these answer set solvers are provided in Section 6.5.

Figures 8.1 and 8.2 present the running times for CMODELS versus SMODELS on Schur Numbers, Putting Numbers, and Blocked n-queens benchmarks respectively. In all tables we include the results already reported in Section 6.5.2 for traditional programs and report new results for new encodings of the problems that contain choice or cardinality constraint rules. We also report grounding times for both encodings. We used the grounder GRINGO in these experiments. Figures 8.1 and 8.2 demonstrate how behavior of answer set solvers may depend on difference in encoding. In Blocked n-queens the difference is especially noticable.

The ASPARAGUS platform also contains benchmarks for the problems *Latin Squares*, *Weighted Latin Squares*, and *Weight-Bounded Dominating Set*. Here are short descriptions of these problems:

- In the *Latin Square* problem we have an n x n board. The task is to assign integers $1, \ldots, n$ to each cell of the board in a way that each integer occurs exactly once in each row and exactly once in each column.

- The *Weighted Latin Square* problem is a variant of Latin Square problem where we are also given n x n weights $wt_{i,j}$ and a bound w. The task is to assign integers $1, \ldots, n$ to the cells of the board in a way that not only each integer occurs exactly once in each row and column, but also that for every row i of the board the following inequality holds

$$a_{i,1} * wt_{i,1} + \cdots + a_{i,n} * wt_{i,n} \leq w$$

  where $a_{i,j}$ is a number assigned to cell (i, j).

- In the *Weight-Bounded Dominating Set* problem we are given a directed graph $G = (V, E)$ where V is the set of vertices, E is the set of edges, and each edge $(i, j) \in E$ is associated with a weight $wt_{(i,j)}$. We are also given a cardinality

k and a weight w. The task is to find a subset D of V such that $|D| \leq k$ and for each vertex $v \in V$ at least one of the following conditions holds:

1. $v \in D$,
2. $\sum_{(i,v)\in E, i\in D} wt_{(i,v)} \geq w$, or
3. $\sum_{(v,j)\in E, j\in D} wt_{(v,j)} \geq w$.

Figure 8.3 presents the running times for CMODELS versus SMODELS on Latin Square, Weighted Latin Square, Weight Bounded Dominating Set benchmarks. The encoding of the first problem contains choice rules and cardinality constrains. The encodings of the other two problems contain choice and weight rules. We use the grounder LPARSE version 1.1.1 to ground all instances in Figure 8.3. We note that both CMODELS using MINISAT and CMODELS using ZCHAFF often outperform SMODELS. In general, using such features as weight and cardinality constraint rules is a big advantage of such systems as SMODELS and CMODELS. First, these constructs often allow us to encode problems in a more elegant and economical way. Second, using such rules usually improves the performance of the systems. Nevertheless, due to the fact that the elimination of weight and cardinality constraint rules may lead to substantial growth of the size of the program, CMODELS may not always benefit from the use of these features in terms of performance.

Encoding	GRINGO		CMODELS + MINISAT		SMODELS	
	tr-l	ext-d	tr-l	ext-d	tr-l	ext-d
color.n100.3	0.03	0.04	0.03	0.02	0.02	0.01
color.n300.3	0.14	0.16	0.1	0.1	0.07	0.08
color.n600.3	0.41	0.44	0.24	0.24	0.16	0.16
color.n1000.3	0.99	1.01	0.43	0.42	0.3	0.28
color.n3000.3	7.56	6.41	1.48	1.4	0.97	0.92
color.n6000.3	33.63	22.74	3.15	2.93	2.0	1.85
color.n100.4	0.04	0.04	0.05	0.04	0.12	0.08
color.n300.4	0.19	0.2	0.16	0.14	1.47	0.71
color.n600.4	0.53	0.53	0.39	0.32	6.72	3.16
color.n1000.4	1.27	1.24	0.74	0.62	21.47	11.43
color.n3000.4	9.91	8.13	8.94	14.1	211.58	109.88
color.n6000.4	44.81	29.87	t-o	t-o	t-o	465.12
schur.p4n43	0.06	0.05	0.15	0.28	0.58	0.18
schur.p4n44	0.06	0.05	0.31	0.9	32.58	0.2
schur.p4n45	0.06	0.05	0.27	4.53	44.3	536.86
schur.p5n100	0.34	0.34	4.51	1.1	t-o	t-o
schur.p5n110	0.41	0.39	16.91	521.08	t-o	t-o
schur.p5n120	0.48	0.46	362.79	t-o	t-o	t-o
schur.p5n130	0.57	0.54	t-o	t-o	t-o	t-o
pn.gsquare-3-10-2-8	0.02	0.02	0.02	0.02	0.06	0.08
pn.gsquare-4-11-3-8	0.04	0.04	0.07	0.15	112.92	57.62
pn.gsquare-4-12-3-8	0.04	0.04	0.07	0.64	6.18	19.05
pn.gsquare-4-14-3-8	0.05	0.05	0.07	0.09	10.61	0.34
pn.gsquare-4-19-3-8	0.07	0.07	0.98	0.14	223.02	223.91
pn.gsquare-4-22-3-8	0.09	0.09	2.48	0.77	105.38	32.08
pn.gsquare-4-24-3-8	0.1	0.1	0.3	0.42	402.48	470.57
pn.gsquare-4-9-3-8	0.03	0.03	0.04	0.03	0.54	1.66
pn.gsquare-5-12-4-8	0.07	0.07	82.4	24.42	t-o	t-o
pn.gsquare-7-25-6-8	0.31	0.28	t-o	t-o	t-o	t-o

Figure 8.1: Graph Coloring, Schur Number, and Putting Numbers; runtimes of GRINGO, CMODELS using MINISAT, and SMODELS on traditional programs and (extended) programs with choice and cardinality constraint rules.

Encoding	GRINGO		CMODELS + MINISAT		SMODELS	
	tr-l	extended	tr-l	ext-d	tr-l	ext-d
queens.18	0.16	0.07	0.22	0.13	6.8	2.09
queens.22	0.3	0.14	0.49	0.29	t-o	171.46
queens.24	0.4	0.19	0.71	0.4	t-o	225.61
queens.28	0.68	0.33	1.26	0.81	t-o	t-o
queens.32	1.08	0.54	2.19	1.5	t-o	t-o
queens.36	1.64	0.83	3.41	2.5	t-o	t-o
bqueens.50.1642398261	5.45	2.82	157.11	19.14	t-o	321.73
bqueens.50.1642399343	5.62	2.8	t-o	7.76	t-o	66.53
bqueens.50.1642399526	5.35	2.8	t-o	24.56	t-o	11.85
bqueens.50.1642400086	5.47	2.82	t-o	20.48	t-o	377.66
bqueens.50.1642401471	5.29	2.82	t-o	5.23	t-o	34.03
bqueens.50.1642402365	5.41	2.8	t-o	49.96	t-o	250.9
bqueens.50.1642402587	5.56	2.76	87.1	3.67	t-o	24.43
bqueens.50.1642403758	5.49	2.81	t-o	21.9	t-o	457.62
bqueens.50.1642404800	5.39	2.81	t-o	4.47	t-o	71.45
bqueens.50.1642405183	5.48	2.77	t-o	35.6	t-o	138.5
bqueens.50.1642405538	5.38	2.81	t-o	0.92	t-o	5.82
bqueens.50.1642405851	5.39	2.81	t-o	1.2	t-o	5.64
bqueens.50.1642406103	5.42	2.8	t-o	30.59	t-o	205.31
bqueens.50.1642406388	5.44	2.8	104.12	8.3	t-o	10.06
bqueens.50.1642406727	5.2	2.8	266.84	1.65	t-o	13.67
bqueens.50.1642407126	5.37	2.8	485.78	5.47	t-o	38.37
bqueens.50.1642407701	5.34	2.8	t-o	12.32	t-o	164.49
bqueens.50.1642407857	5.47	2.8	119.3	2.36	t-o	25.87
bqueens.50.1642408305	5.44	2.8	t-o	26.07	t-o	184.88
bqueens.50.1642408561	5.4	2.8	201.16	50.75	t-o	11.52

Figure 8.2: n-queens and Blocked n-queens problem; runtimes of GRINGO, CMODELS using MINISAT, and SMODELS on traditional programs and (extended) programs with choice rules.

Encoding	LPARSE	CMODELS		SMODELS
		MINISAT	ZCHAFF	
latinsquare.1706819821	3.14	7.94	10.76	t-o
latinsquare.1706819916	3.15	7.89	10.02	t-o
latinsquare.1706821187	3.12	7.9	6.94	t-o
latinsquare.1706821345	3.21	7.8	26.11	17.21
latinsquare.1706821885	3.16	7.86	9.37	11.43
latinsquare.1706823705	3.21	7.77	6.88	211.59
latinsquare.1706823818	3.14	7.82	6.74	228.04
latinsquare.1706823943	3.12	7.81	6.58	8.14
latinsquare.1706823984	3.14	7.87	7.95	9.45
latinsquare.1706824107	3.13	7.86	11.0	83.08
weightedls.1162362547	0.04	0.57	0.4	0.02
weightedls.1162368434	0.04	0.56	0.29	513.44
weightedls.1162378470	0.04	0.55	0.25	133.41
weightedls.1162549540	0.05	0.61	0.52	58.87
weightedls.1162571546	0.04	0.55	0.25	68.02
weightedls.1162579118	0.04	0.51	0.38	7.06
weightedls.1162583044	0.04	0.58	0.32	0.02
weightedls.1162586028	0.04	0.58	0.26	145.08
weightedls.1162588733	0.04	0.64	0.9	101.07
weightedls.1162592956	0.04	0.56	0.26	0.01
rand-100-400-1159666138-13	0.09	82.57	12.15	t-o
rand-100-400-1159666138-19	0.09	0.21	12.96	t-o
rand-100-400-1159666138-1	0.09	0.46	25.98	209.38
rand-100-400-1159666138-2	0.09	t-o	57.04	17.57
rand-100-400-1159666138-3	0.08	43.32	6.47	t-o
rand-150-600-1159731678-11	0.16	572.6	33.92	t-o
rand-150-600-1159731678-12	0.16	t-o	135.27	t-o
rand-150-600-1159731678-14	0.16	10.36	269.31	t-o
rand-150-600-1159731678-3	0.16	54.78	t-o	t-o
rand-150-600-1159731678-5	0.16	217.73	175.04	t-o
rand-200-800-1159728969-1	0.24	559.73	t-o	t-o
rand-200-800-1159728969-4	0.24	345.94	t-o	t-o
rand-200-800-1159728969-7	0.24	228.64	t-o	t-o
rand-200-800-1159728969-8	0.24	89.11	t-o	t-o

Figure 8.3: Latin Square, Weighted Latin Square, Weight Bounded Dominating Set; runtimes of LPARSE, CMODELS using MINISAT and ZCHAFF, and SMODELS on programs with choice and weight rules.

Chapter 9

Background: Loop Formulas

As discussed in Section 7.6, for any semi-traditional program, each answer set is a model of the program's completion, but the converse is in general not true unless the program is assumed to be tight. Investigation of techniques that allow us to use satisfiability solvers for computing answer sets of arbitrary, possibly nontight, logic programs is the main topic of the rest of this dissertation. This chapter introduces an important concept used in this research, the notion of a loop formula.

9.1 Loop Formula

Lin and Zhao [2002] showed that if we extend the completion of a traditional program by "loop formulas" then models of the resulting formula will be identical to the answer sets of this program. Lee and Lifschitz [2003] extended their results to the case of nested programs. In this section we review these findings for the case of semi-traditional programs. Recall that rules of a semi-traditional program have the form

$$a \leftarrow D, F$$

where a is an atom or $\bot$, D stands for

$$b_1, \ldots, b_l,$$

and F stands for the expression

$$not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n,$$

and each b_i is an atom.

A nonempty set X of atoms is called a *loop*[1] of a semi-traditional program Π if, for every pair a and b of atoms in X, there exists a path (possibly of length 0) from a to b in the dependency graph of Π (see Section 7.6) such that all vertices in this path belong to X. In other words, X is a loop of Π if and only if the subgraph of the dependency graph of Π induced by X is strongly connected. It is clear that any set consisting of a single atom is a loop.

For instance, consider a program

$$\begin{aligned} a &\leftarrow\ not\ d \\ a &\leftarrow c \\ b &\leftarrow c \\ c &\leftarrow a, b. \end{aligned} \tag{9.1}$$

The dependency graph of the program is shown in Figure 9.1. The program has seven loops: $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$. The first four loops are trivial – they are singletons corresponding to the atoms occurring in the program.

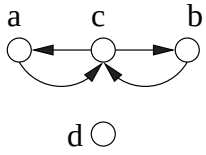

Figure 9.1: Dependency graph of program (9.1)

The loop formulas of Π are determined by the rules of Π whose head is an atom, rather than $\bot$:

$$a \leftarrow D, F \tag{9.2}$$

The *loop formula* F_L of a loop L is

$$\bigvee L \rightarrow \bigvee R(L) \tag{9.3}$$

[1]Definitions of loop in [Lin and Zhao, 2002] and [Lee and Lifschitz, 2003] differ. We follow the tradition of the latter.

Loop	Loop Formula
$\{a\}$	$a \rightarrow \neg d \vee c$
$\{b\}$	$b \rightarrow c$
$\{c\}$	$c \rightarrow a \wedge b$
$\{d\}$	$d \rightarrow \bot$
$\{a, c\}$	$a \vee c \rightarrow \neg d$
$\{b, c\}$	$b \vee c \rightarrow \bot$
$\{a, b, c\}$	$a \vee b \vee c \rightarrow \neg d$

Figure 9.2: Loops and loop formulas for program (9.1).

where $R(L)$ is the set of formulas

$$D \wedge F \tag{9.4}$$

for all rules (9.2) in Π such that $a \in L$ and $D \cap L = \emptyset$. (By $\bigvee L$ we denote the disjunction of all elements of L, and $\bigvee R(L)$ is understood in a similar way). By $LF(\Pi)$ we denote the set (the conjunction) of all loop formulas for Π.

For instance, the loop formulas for program (9.1) are shown in Figure 9.2. $LF(\Pi)$ is the set of formulas in the right column.

It is interesting to note that for a singleton loop, the corresponding set of the bodies of some rules form the right hand side of the implication (4.10) in the program's completion. (In fact, if there is no rule of the form $a \leftarrow \ldots, a, \ldots$ in the program then the loop formula of $\{a\}$ is identical to implication (4.10) of the completion for atom a). By $SLF(\Pi)$ we denote the set of loop formulas of all singletons for Π. It is clear that $\Pi \cup SLF(\Pi)$ entails the program's completion $Comp(\Pi)$. Since $SLF(\Pi)$ is a subset of $LF(\Pi)$, $\Pi \cup LF(\Pi)$ entails $Comp(\Pi)$ as well.

For instance, consider Π

$$a \leftarrow a.$$

Its completion $Comp(\Pi)$ is identical to Π. On the other hand, $\Pi \cup SLF(\Pi)$ is

$$a \rightarrow a$$
$$a \rightarrow \bot$$

where the last formula is the loop formula of the singleton loop $\{a\}$.

Let Π be program (9.1). Set $SLF(\Pi)$ consists of the first four loop formulas shown in Figure 9.2. In this case, the union $\Pi \cup SLF(\Pi)$

$$\begin{array}{l} \neg d \rightarrow a \\ c \rightarrow a \\ c \rightarrow b \\ a \wedge b \rightarrow c \\ a \rightarrow \neg d \vee c \\ b \rightarrow c \\ c \rightarrow a \wedge b \\ d \rightarrow \bot \end{array} \tag{9.5}$$

is identical to the completion $Comp(\Pi)$.

Since all loops of any tight program Π are singletons, the union of a tight program with the set of its loop formulas $\Pi \cup LF(\Pi)$ is always identical to the completion $Comp(\Pi)$.

The following theorem is a special case of Theorem 1 from [Lee and Lifschitz, 2003].

Theorem 11. *For any semi-traditional program* Π *and any set* X *of atoms,* X *is an answer set for* Π *if and only if* X *satisfies* $\Pi \cup LF(\Pi)$.

In a sense, $\Pi \cup LF(\Pi)$ is an "improved" version of $Comp(\Pi)$.

For instance, let Π be again program (9.1). This program has six models $\{a\}$, $\{d\}$, $\{a, d\}$, $\{b, d\}$, $\{a, b, c\}$, $\{a, b, c, d\}$. Nevertheless, only one model $\{a\}$ satisfies all of the program's loop formulas (see Figure 9.2). Therefore, the program has only one answer set $\{a\}$.

On the other hand, let us consider the program's completion (9.5). The completion has two models $\{a\}$, $\{a, b, c\}$. The model $\{a, b, c\}$ does not satisfy the loop formula $b \vee c \rightarrow \bot$ of the program, and hence, it is not an answer set.

We call formula $\Pi \cup LF(\Pi)$ Lin-Zhao transformation of a program Π. Answer sets of any semi-traditional logic program coincide with models of its Lin-Zhao transformation. In other words, a set X of literals is a model of $\Pi \cup LF(\Pi)$ if and only if X^+ is an answer set of Π. The question that comes into mind is: How can we use Lin-Zhao transformation for finding answer sets of a nontight program?

Consider a program Π. To find the answer sets of Π, one possibility is to

(1) compute the set $\Gamma = \Pi \cup LF(\Pi)$ of formulas, and then

(2) invoke a SAT solver to determine the models of Γ.

This is an "eager"approach which may work well in some cases, but in general the resulting propositional formula $\Pi \cup LF(\Pi)$ may be exponentially bigger than the input program Π.

For instance, consider a program consisting of n^2 rules

$$a_i \leftarrow a_j, \quad (1 \leq i, j \leq n).$$

Since the dependency graph of this program is complete, every nonempty subset of atoms in the program forms a loop. Therefore there are $2^n - 1$ loop formulas for this program.

More generally a result by Lifschitz and Razborov [2006] shows that — assuming $P \not\subseteq NC^1/poly$, a conjecture from computational complexity theory widely believed to be true — whenever we try to translate a logic program into an equivalent set of propositional formulas, an exponential blow up may occur.

For this reason, the straightforward use of the Lin-Zhao transformation is, generally, not feasible. Next section reviews the SAT-based method used in the system ASSAT [Lin and Zhao, 2002] that utilizes the concept of a loop formula for finding answer sets of a nontight program using a "lazy" approach. Unlike the "eager" approach, which involves computing all loop formulas of the program at once, the "lazy" methodology performs the computation of loop formulas on demand. In many cases, this leads to avoiding the exponential blow up and makes the SAT-based approach applicable to many large nontight programs.

9.2 SAT-based System Assat

ASSAT [Lin and Zhao, 2002; 2004] is a SAT-based system for traditional programs that takes a "lazy" approach to the use of the Lin-Zhao transformation. Instead of computing all loop formulas at once, ASSAT adds loop formulas on demand, i.e., ASSAT

1. Computes $\Gamma = Comp(\Pi)$.

2. Looks for a model X of Γ using a SAT solver; if no such model exists then the input program does not have answer sets and the procedure terminates returning *False*;

3. Checks if X is an answer set; if yes, then the procedure terminates returning *True*; otherwise, ASSAT

   (a) finds a loop formula that is not satisfied by X, and adds it to Γ;

   (b) goes back to step 2.

The method for computing loop formulas used in Step 3a is one of the important contributions of [Lin and Zhao, 2002]. (We discuss this method extended to semi-traditional programs in Section 11.3). The above procedure can easily be modified for finding all answer sets of the program, rather than just one. Lin and Zhao [2002; 2004] showed that ASSAT can often outperform specialized answer set solvers such as, for instance, SMODELS. However, ASSAT has the following two drawbacks:

1. In two successive calls to the SAT solver, the computation performed for finding the first model is completely discarded, i.e., not re-used by the SAT solver in the second call. Thus some branches of the search tree may be computed many times.

2. ASSAT is not guaranteed to work in polynomial space due to the fact that there are programs for which there are exponentially many loop formulas that cannot be derived from the program's completion and from other loop formulas. Consider what happens when ASSAT is applied to such a program Π:

   - If Π has an answer set, then the performance of ASSAT on Π depends on how lucky the system is in generating the right model early. In the best case it generates an answer set quickly. In the worst case it blows up in space.
   - If Π has no answer set, then ASSAT blows up in space. In fact, adding and keeping already added loop formulas is essential to guarantee that the SAT solver does not return an already computed model, and thus to guarantee termination.

In the next chapter we discuss an alternative SAT-based approach that resolves these two problems.

Chapter 10

Abstract Description of "Generate and Test" DPLL

In this chapter we will define a modification of the graph underlying DPLL that includes testing assignments found by DPLL. In the next chapter we will show how such graph can be used to extend the CMODELS algorithm to nontight programs. In Section 10.1 we present a graph $\mathrm{GT}_{F,G}$ that is a modification of the DPLL graph DP_F (Section 3.2) which includes the testing assignments of F found by DPLL. This graph describes a basic "generate and test" algorithm based on DPLL. In Section 10.2 we extend the graph $\mathrm{GT}_{F,G}$ to $\mathrm{GTL}_{F,G}$ to capture backjumping and learning for the generate and test algorithm. Section 10.3 defines the "extended" graph $\mathrm{GTL}^{\uparrow}_{F,G}$ that adopts more detailed notion of a state. Such extended graph is better suited for formalizing ideas behind computing backjump clauses used in conflict driven backjumping and learning. We use $\mathrm{GTL}^{\uparrow}_{F,G}$ in Sections 10.4 and 10.5 to define the procedures *BackjumpClause* and *BackjumpClauseFirstUIP* for computing *Decision* and *FirstUIP* backjump clauses respectively. Section 10.4 also provides proofs of the theorems stated in earlier sections of the chapter.

10.1 Abstract Generate and Test

Recall that any answer set of a program is also a model of its completion. It follows that DPLL — an algorithm for finding the models of a propositional formula — applied to the clausified completion of a program enumerates a set containing

all answer sets of the program. If we modify the DPLL algorithm to allow testing assignments, we can use the resulting algorithm to identify only those models of completion that are also answer sets. We call such an approach *generate* and *test*.

In [Giunchiglia *et al.*, 2006], we defined an ASP-SAT algorithm that utilizes the generate and test approach for computing answer sets of a program. Given a program Π, ASP-SAT generates the models of *ED-Comp*(Π) — the completion converted to CNF by means of auxiliary variables (Section 6.3) — using DPLL, and tests them until an answer set is found.

In this section, we present a modification of the graph DP_F (Section 3.2) that includes testing assignments of F found by DPLL. This modification of the graph DP_F is of interest because it can be used to describe the ASP-SAT algorithm. A new graph $\text{GT}_{F,G}$ is such that if F is the completion of a program Π and G is $LF(\Pi)$ then the terminal nodes of this graph correspond to the program's answer sets. In Section 11.1 we use the graph $\text{GT}_{F,G}$ to describe the ASP-SAT algorithm.

Let F be a CNF formula, and let G be a formula formed from atoms occurring in F. The terminal nodes of the graph $\text{GT}_{F,G}$ defined below are models of formula $F \wedge G$.

The nodes of the graph $\text{GT}_{F,G}$ are the same as the nodes of the graph DP_F. The edges of $\text{GT}_{F,G}$ are described by the transition rules of DP_F and the additional transition rule:

Test:

$$M \implies M\ \overline{l} \text{ if } \begin{cases} M \text{ is consistent,} \\ G \models \overline{M}, \\ l \in M \end{cases}$$

It is easy to see that the graph DP_F is a subgraph of $\text{GT}_{F,G}$. The latter graph can be used for deciding whether a formula $F \wedge G$ has a model by constructing a path from $\emptyset$ to a terminal node:

Theorem 12. *For any CNF formula F and a formula G formed from atoms occurring in F,*

(a) graph $\text{GT}_{F,G}$ is finite and acyclic,

(b) any terminal state of $\text{GT}_{F,G}$ other than FailState is a model of $F \wedge G$,

(c) FailState is reachable from $\emptyset$ in $\text{GT}_{F,G}$ if and only if $F \wedge G$ is unsatisfiable.

Note that to verify the applicability of the new transition rule *Test* we need a procedure for testing whether G entails a clause, but there is no need to explicitly write out G. This is important because $LF(\Pi)$ can be very long [Lin and Zhao, 2002].

As discussed in Section 9.1, answer sets of a program Π can be characterized as models of its completion extended by so called loop formulas of a program. Recall that *ED-Comp*(Π) is the completion converted to CNF using auxiliary variables (Section 6.3) and $LF(\Pi)$ is the conjunction of all loop formulas of Π (Section 9.1). For a set M of literals by M_Π we denote all literals in M whose atoms occur in Π. For instance, let Π contain two atoms a and b, and M be $a\ \neg b\ c$ then M_Π is $a\ \neg b$. It is easy to see that for any program Π and any assignment M of *ED-Comp*$(\Pi) \wedge LF(\Pi)$, M is a model of *ED-Comp*$(\Pi) \wedge LF(\Pi)$ if and only if M_Π^+ is an answer set of Π.

Let Π be the nontight program

$$d \leftarrow d.$$

Its completion is

$$d \leftrightarrow d,$$

and *ED-Comp*(Π) is

$$(d \vee \neg d).$$

This program has one loop formula

$$d \rightarrow \bot.$$

Theorem 12 asserts that a terminal state $\neg d$ of $\mathrm{GT}_{\textit{ED-Comp}(\Pi),\ d\rightarrow\bot}$ is a model of *ED-Comp*$(\Pi) \wedge LF(\Pi)$. It follows that $\{\neg d\}^+ = \emptyset$ is an answer set of Π. To compare $\mathrm{GT}_{\textit{ED-Comp}(\Pi),\ d\rightarrow\bot}$ with the graph $\mathrm{DP}_{\textit{ED-Comp}(\Pi)}$: state d is a terminal state in $\mathrm{DP}_{\textit{ED-Comp}(\Pi)}$ whereas d is not a terminal state in $\mathrm{GT}_{\textit{ED-Comp}(\Pi),\ d\rightarrow\bot}$ because the transition rule *Test* is applicable to this state.

In the rest of this section we give a proof of Theorem 12.

Lemma 7. *For any CNF formula F, a formula G formed from atoms occurring in F, and a path from $\emptyset$ to a state $l_1\ \ldots\ l_n$ in $\mathrm{GT}_{F,G}$, any model X of $F \wedge G$ satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.*

Proof. By induction on the path from $\emptyset$ to $l_1 \ \ldots \ l_n$. Similar to the proof of Lemma 1. We will show that the property in question is preserved by the transition rule *Test*.

Take a model X of $F \wedge G$ and consider an edge $M \Longrightarrow M'$ where M is a list $l_1 \ldots l_k$ such that X satisfies l_i if it satisfies all decision literals l_j^{Δ} with $j \leq i$.

Assume that X satisfies all decision literals from M'. We will show that the rule justifying the transition from M to M' is different from *Test*. By contradiction. M' is $M\ \overline{l}$. From the assumption that X satisfies all decision literals from M', it follows that X satisfies all decision literals from M. By the inductive hypothesis, $X \models M$. By the definition of *Test*, $G \models \overline{M}$. Since X is a model of $F \wedge G$ it follows that $X \models \overline{M}$. This contradicts the fact that $X \models M$. □

Proof of Theorem 12

Part (a) and part (c) Right-to-left are proved as in the proof of Theorem 1.

(b) Let M be any terminal state other than *FailState*. As in the proof of Theorem 1(b) it follows that M is a model of F. The transition rule *Test* is not applicable. Hence $G \not\models \overline{M}$. In other words M is a model of G. We conclude that M is a model of $F \wedge G$

(c) Left-to-right: Since *FailState* is reachable from $\emptyset$, there is a state M without decision literals such that M is reachable from $\emptyset$ and the transition rule *Fail* is applicable in M. Then, M is inconsistent. By Lemma 7, any model of $F \wedge G$ satisfies M. Since M is inconsistent we conclude $F \wedge G$ is unsatisfiable. □

10.2 Abstract Generate and Test with Backjumping and Learning

In this section we model backjumping and learning techniques for the generate and test procedure introduced in previous section by defining a graph $\text{GTL}_{F,G}$ that extends $\text{GT}_{F,G}$ in a similar way as DPL_F extends DP_F (see Sections 3.2 and 3.4).

An *(augmented) state* relative to a CNF formula F and a formula G formed from atoms occurring in F is either a distinguished state *FailState* or a pair of the form $M||\Gamma$, where M is a record (Section 3.2) relative to the set of atoms occurring in F, and Γ is a (multi)set of clauses formed from atoms occurring in F that are entailed by $F \wedge G$.

The nodes of the graph $\text{GTL}_{F,G}$ are the augmented states relative to a CNF formula F and a formula G formed from atoms occurring in F. The edges of $\text{GTL}_{F,G}$ are described by the transition rules *Unit Propagate* λ, *Decide*, *Fail* of DPL_F, the transition rules

$$\begin{array}{l}
\textit{Backjump GT}: \\
P\ l^{\Delta}\ Q||\Gamma \Longrightarrow\ P\ l'||\Gamma\ \text{ if }\ \left\{\begin{array}{l} P\ l^{\Delta}\ Q \text{ is inconsistent and} \\ F \wedge G \models l' \vee \overline{P} \end{array}\right. \\
\textit{Learn GT}: \\
M||\Gamma\ \Longrightarrow\ M||\ C,\ \Gamma\ \text{ if }\ \left\{\begin{array}{l} \text{every atom in } C \text{ occurs in } F \text{ and} \\ F \wedge G \models C \end{array}\right.
\end{array}$$

and the transition rule *Test* of $\text{GT}_{F,G}$ that is extended to $\text{GTL}_{F,G}$ as follows: $M||\Gamma \Longrightarrow M'||\Gamma$ is an edge in $\text{GTL}_{F,G}$ justified by *Test* if and only if $M \Longrightarrow M'$ is an edge in $\text{GT}_{F,G}$ justified by *Test*.

We refer to the transition rules *Unit Propagate* λ, *Test*, *Decide*, *Fail*, *Backjump GT* of the graph $\text{GTL}_{F,G}$ as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn GT* is applicable to it.

The graph $\text{GTL}_{F,G}$ can be used for deciding whether a formula $F \wedge G$ has a model by constructing a path from $\emptyset||\emptyset$ to a terminal node:

Theorem 13. *For any CNF formula F and a formula G formed from atoms occurring in F,*

(a) every path in $\text{GTL}_{F,G}$ contains only finitely many edges labeled by Basic transition rules,

(b) for any semi-terminal state $M||\Gamma$ of $\text{GTL}_{F,G}$ reachable from $\emptyset||\emptyset$, M is a model of $F \wedge G$,

(c) FailState is reachable from $\emptyset||\emptyset$ in $\text{GTL}_{F,G}$ if and only if $F \wedge G$ is unsatisfiable.

Thus if we construct a path from $\emptyset||\emptyset$ so that Basic transition rules periodically appear in it then some semi-terminal state will be eventually reached; as soon as a semi-terminal state is reached the problem of finding a model of $F \wedge G$ is solved.

For instance, let Π be nontight program (7.4). Its completion is

$$(a \leftrightarrow a) \wedge (b \leftrightarrow \neg a)$$

and *ED-Comp*(Π) is

$$(a \vee \neg a) \wedge (a \vee b) \wedge (\neg a \vee \neg b).$$

This program has one loop formula

$$a \rightarrow \bot.$$

Here is a path in $\text{GTL}_{\textit{ED-Comp}(\Pi),a\rightarrow\bot}$:

$$\begin{array}{ll}
\emptyset||\emptyset & \Longrightarrow \ (\textit{Decide}) \\
a^{\Delta}||\emptyset & \Longrightarrow \ (\textit{Unit Propagate } \lambda) \\
a^{\Delta}\ \neg b||\emptyset & \Longrightarrow \ (\textit{Test}) \\
a^{\Delta}\ \neg b\ \neg a||\emptyset & \Longrightarrow \ (\textit{Backjump GT}) \\
\neg a||\emptyset & \Longrightarrow \ (\textit{Unit Propagate } \lambda) \\
\neg a\ \neg b||\emptyset &
\end{array} \tag{10.1}$$

Since the state $\neg a\ \neg b$ is semi-terminal, Theorem 13 (b) asserts that $\{\neg a, \neg b\}$ is a model of *ED-Comp*(Π) ∧ $(a \rightarrow \bot)$.

As in case of the graph DPL_F, the transition rule *Backjump GT* is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$. We call such states *backjump* states.

Theorem 14. *For any CNF formula F and a formula G formed from atoms occurring in F, the transition rule Backjump GT is applicable in any backjump state in* $\text{GTL}_{F,G}$.

Proofs of Theorems 13 and 14 are given in Section 10.4.

10.3 Backjumping and Extended Graph

Recall the transition rule *Backjump GT* of $\text{GTL}_{F,G}$

$$\begin{array}{l}
\textit{Backjump GT}: \\
P\ l^{\Delta}\ Q||\Gamma \Longrightarrow\ P\ l'||\Gamma \ \text{ if } \ \left\{ \begin{array}{l} P\ l^{\Delta}\ Q \text{ is inconsistent and} \\ F \wedge G \models l' \vee \overline{P} \end{array} \right.
\end{array}$$

A state in the graph $\text{GTL}_{F,G}$ is a *backjump state* if it is inconsistent, contains a decision literal, and is reachable from $\emptyset||\emptyset$. Note that it may not be clear a priori whether *Backjump GT* is applicable to a backjump state and if so to which state the edge due to the application of *Backjump GT* leads. These questions are important if we want to base an algorithm on this framework. Theorem 14 (Section 10.2) asserts that *Backjump GT* is always applicable to a backjump state, so that a backjump state in $\text{GTL}_{F,G}$ is never semi-terminal. In the end of this section we show how Theorem 14 can be derived from the results proved later in this dissertation. The next question to answer is how to continue choosing a path in the graph after reaching a backjump state. To answer this question we introduce the notions of a reason and an extended graph.

For a formula H, we say that a clause $l \vee C$ is a *reason* for l to be in a list $P\ l\ Q$ of literals with respect to H if $H \models l \vee C$ and $\overline{C} \subseteq P$. We can equivalently restate the second condition of *Backjump GT* "$F \wedge G \models l' \vee \overline{P}$" as "there exists a reason for l' to be in $P\ l'$ with respect to $F \wedge G$" (note that $l' \vee \overline{P}$ is a reason for l' to be in $P\ l'$). We call a reason for l' to be in $P\ l'$ a *backjump clause*. Note that Theorem 14 asserts that a backjump clause always exists for a backjump state. It is clear that we may continue choosing a path in the graph after reaching a backjump state if we know how to compute a backjump clause for this state. We now define a graph $\text{GTL}^{\uparrow}_{F,G}$ that shares many properties of $\text{GTL}_{F,G}$ but allows us to give a simpler procedure for computing a backjump clause.

An *(extended) record* M relative to a formula H is a list of literals over the set of atoms occurring in H where

(i) each literal l in M is annotated either by Δ or by a reason for l to be in M with respect to H,

(ii) M contains no repetitions,

(iii) for any inconsistent prefix of M, its last literal is annotated by a reason.

For instance, let H be a formula

$$(a \vee b)\ \wedge\ (\neg b \vee \neg a)\ \wedge\ c.$$

The following lists of literals

$$b^{\Delta}\ a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}, \quad b^{\Delta}\ \neg\ a^{\neg b \vee \neg a}$$

are extended records relative to H. On the other hand, the lists of literals

$$a^{\Delta}\ \neg a^{\Delta}, \qquad a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}\ b^{\Delta}, \qquad b^{\Delta}\ a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}\ c^{\Delta}$$

are not extended records.

An *(extended) state* relative to a CNF formula F, and a formula G formed from atoms occurring in F is either the distinguished state *FailState* or a pair of the form $M||\Gamma$, where M is an extended record relative to $F \wedge G$, and Γ is the same as in the definition of an augmented state (i.e., Γ is a (multi)set of clauses formed from atoms occurring in F that are entailed by $F \wedge G$.) It is easy to see that for any extended state S relative to F and G, the result of removing annotations from all nondecision literals of S is a state of $\mathrm{GTL}_{F,G}$: we will denote this state by $S^{\downarrow}$.

For instance, let formula F be $a \vee b$ and G be $\top$. All pairs

$$\mathit{FailState} \quad \emptyset||\emptyset \qquad \neg a^{\Delta}\ b^{b \vee a}||\emptyset \qquad \neg b^{\Delta}\ a^{a \vee b}||\emptyset$$

are among valid extended states relative to these formulas. The corresponding states $S^{\downarrow}$ are

$$\mathit{FailState} \qquad \emptyset||\emptyset \qquad \neg a^{\Delta}\ b||\emptyset \qquad \neg b^{\Delta}\ a||\emptyset.$$

We now define a graph $\mathrm{GTL}^{\uparrow}_{F,G}$ for any CNF formula F and any formula G formed from atoms occurring in F. The set of the nodes of $\mathrm{GTL}^{\uparrow}_{F,G}$ consists of the extended states relative to F and G. The transition rules of $\mathrm{GTL}_{F,G}$ are extended to $\mathrm{GTL}^{\uparrow}_{F,G}$ as follows: $S_1 \implies S_2$ is an edge in $\mathrm{GTL}^{\uparrow}_{F,G}$ justified by a transition rule T if and only if $S_1^{\downarrow} \implies S_2^{\downarrow}$ is an edge in $\mathrm{GTL}_{F,G}$ justified by T.

The definitions of Basic transition rules and semi-terminal states in $\mathrm{GTL}^{\uparrow}_{F,G}$ are similar to their definitions for $\mathrm{GTL}_{F,G}$.

Theorem 13$^{\uparrow}$. *For any CNF formula F and a formula G formed from atoms occurring in F,*

(a) every path in $\mathrm{GTL}^{\uparrow}_{F,G}$ contains only finitely many edges labeled by Basic transition rules,

(b) for any semi-terminal state $M||\Gamma$ of $\mathrm{GTL}^{\uparrow}_{F,G}$, M is a model of $F \wedge G$,

(c) $\mathrm{GTL}^{\uparrow}_{F,G}$ contains an edge leading to FailState if and only if $F \wedge G$ is unsatisfiable.

Note that Theorem $13^{\uparrow}$ (b), unlike Theorem 13 (b), does not require a semi-terminal state to be reachable from $\emptyset||\emptyset$. As in the case of the graph $\mathrm{GTL}_{F,G}$, $\mathrm{GTL}^{\uparrow}_{F,G}$ can be used for deciding whether a formula $F \wedge G$ has a model. Furthermore, the new graph provides the means for computing a backjump clause that permits practical application of the transition rule *Backjump GT*: Sections 10.4.3 and 10.5 describe the *BackjumpClause* (Algorithm 3) and *BackjumpClauseFirstUIP* (Algorithm 4) procedures that compute *Decision* and *FirstUIP* backjump clauses respectively.

We say that a state in the graph $\mathrm{GTL}^{\uparrow}_{F,G}$ is a *backjump state* if its record is inconsistent and contains a decision literal. Unlike the definition of a backjump state in $\mathrm{GTL}_{F,G}$, this definition does not require a backjump state to be reachable from $\emptyset||\emptyset$ in $\mathrm{GTL}^{\uparrow}_{F,G}$. As in case of the graph $\mathrm{GTL}_{F,G}$, any backjump state in $\mathrm{GTL}^{\uparrow}_{F,G}$ is not semi-terminal:

Theorem $14^{\uparrow}$. *For any CNF formula F and a formula G formed from atoms occurring in F, the transition rule Backjump GT is applicable to any backjump state in $\mathrm{GTL}^{\uparrow}_{F,G}$.*

The lemma below formally states the relationship between nodes of the graphs $\mathrm{GTL}_{F,G}$ and $\mathrm{GTL}^{\uparrow}_{F,G}$:

Lemma 8. *For any CNF formula F and a formula G formed from atoms occurring in F, if S' is a state reachable from $\emptyset||\emptyset$ in the graph $\mathrm{GTL}_{F,G}$ then there is a state S in the graph $\mathrm{GTL}^{\uparrow}_{F,G}$ such that $S^{\downarrow} = S'$.*

Theorems 13 (b), (c), and 14 easily follow from Lemma 8, Theorems $13^{\uparrow}$ (b), (c), and $14^{\uparrow}$ respectively. The proof of Theorem 13 (a) is similar to the proof of Theorem $13^{\uparrow}$ (a).

Section 10.4 will present the proofs for Theorem $13^{\uparrow}$, Lemma 8, and Theorem $14^{\uparrow}$. It is interesting to note that the proofs of Lemma 8 and Theorem $14^{\uparrow}$ implicitly provide the means for following a path in the graph $\mathrm{GTL}^{\uparrow}_{F,G}$:

- given a state $M||\Gamma$ and a transition rule *Unit Propagate* λ, *Test* applicable to $M||\Gamma$, the proof of Lemma 8 describes a clause that may be used to construct

a record M' so that there is an edge $M||\Gamma \implies M'||\Gamma$ due to this transition rule,

- given a backjump state $M||\Gamma$, the proof of Theorem $14^{\uparrow}$ describes a backjump clause that can be used to construct a record M' so that there is an edge $M||\Gamma \implies M'||\Gamma$ due to *Backjump GT*.

Furthermore, the construction of the proof of Theorem $14^{\uparrow}$ paves the way for the procedure *BackjumpClause* presented in Algorithm 3.

10.4 Proofs of Theorem $13^{\uparrow}$, Lemma 8, and Theorem $14^{\uparrow}$

10.4.1 Proof of Theorem $13^{\uparrow}$

Lemma 9. *For any CNF formula F, a formula G formed from atoms occurring in F, an extended record M relative to $F \wedge G$, and any model X of $F \wedge G$, if X satisfies all decision literals in M then $X \models M$.*

Proof. By induction on the length of M. The property trivially holds for $\emptyset$. We assume that the property holds for any state with n elements. Consider any state M with $n+1$ elements. Let X be a model of $F \wedge G$ such that X satisfies all decision literals in M.

Case 1. M has the form $P\, l^{\Delta}$. By the inductive hypothesis, $X \models P$. Since X satisfies all decision literals in M, $X \models l^{\Delta}$.

Case 2. M has the form $P\, l^{l \vee C}$. By the inductive hypothesis, $X \models P$. By the definition of a reason (i) $F \wedge G$ entails $l \vee C$ and (ii) $\overline{C} \subseteq P$. From (ii) it follows that $P \models \neg C$. Consequently, $X \models \neg C$. From (i) it follows that $X \models l \vee C$. We conclude that $X \models l$. □

The proof of Theorem $13^{\uparrow}$ assumes the correctness of Theorem $14^{\uparrow}$ that we demonstrate later in Section 10.4.3.

Theorem $13^{\uparrow}$. *For any CNF formula F and a formula G formed from atoms occurring in F,*

(a) every path in $\mathrm{GTL}^{\uparrow}_{F,G}$ contains only finitely many edges labeled by Basic transition rules,

(b) for any semi-terminal state $M||\Gamma$ *of* $\text{GTL}^{\uparrow}_{F,G}$*,* M *is a model of* $F \wedge G$*,*

(c) $\text{GTL}^{\uparrow}_{F,G}$ *contains an edge leading to FailState if and only if* $F \wedge G$ *is unsatisfiable.*

Proof. (a) For any list N of literals by $|N|$ we denote the length of N. Any state $M||\Gamma$ has the form $M_0\ l_1^{\Delta}\ M_1 \dots l_p^{\Delta}\ M_p||\Gamma$, where $l_1^{\Delta} \dots l_p^{\Delta}$ are all desicion literals of M; we define $\alpha(M||\Gamma)$ as the sequence of nonnegative integers $|M_0|, |M_1|, \dots, |M_p|$, and $\alpha(\mathit{FailState}) = \infty$. For any states S and S' of $\text{GTL}^{\uparrow}_{F,G}$, we understand $\alpha(S) < \alpha(S')$ as the lexicographical order. We first note that for any state $M||\Gamma$, the value of α is based only on the first component M of the state. Second, there is a finite number of distinct values of α due to the fact that there is a finite number of distinct Ms over $F \wedge G$. We derive that there is a finite number of distinct values of α for the states of $\text{GTL}^{\uparrow}_{F,G}$, even though the number of distinct states in $\text{GTL}^{\uparrow}_{F,G}$ is infinite.

By the definition of the transition rules of $\text{GTL}^{\uparrow}_{F,G}$, if there is an edge from $M||G$ to $M'||G'$ in $\text{GTL}^{\uparrow}_{F,G}$ formed by any Basic transition rule then $\alpha(M||G) < \alpha(M'||G')$. Then, due to the fact that there is a finite number of distinct values of α, it follows that there is only a finite number of edges due to the application of Basic rules possible in any path.

(b) Let $M||\Gamma$ be a semi-terminal state so that none of the Basic rules are applicable. From the fact that *Decide* is not applicable, we conclude that M assigns all literals.

Furthermore, M is consistent. Indeed, assume that M is inconsistent. Then, since *Fail* is not applicable, M contains a decision literal. Consequently, $M||\Gamma$ is a backjump state. By Theorem $14^{\uparrow}$, the transition rule *Backjump GT* is applicable in $M||\Gamma$. This contradicts our assumption that $M||\Gamma$ is semi-terminal.

Also, M is a model of F: since *Unit Propagate* λ is not applicable, it follows that for every clause $C \vee l \in F \cup \Gamma$ if $\overline{C} \subseteq M$ then $l \in M$. Consequently, $M \models C \vee l$. Furthermore, M is a model of G: since *Test* is not applicable, then $G \not\models \overline{M}$. We conclude that $M \models G$. Consequently, M is a model of $F \wedge G$.

(c) Left-to-right: There is a state $M||\Gamma$ in $\text{GTL}^{\uparrow}_{F,G}$ such that there is an edge between $M||\Gamma$ and *FailState*. By the definition of $\text{GTL}^{\uparrow}_{F,G}$, this edge is due to the transition rule *Fail*. Consequently, the state $M||\Gamma$ is such that M is inconsistent and contains no decision literals. By Lemma 9, for every X that is a model of $F \wedge G$, $X \models M$. Since M is inconsistent we conclude that $F \wedge G$ has no models.

Right-to-left: Consider the process of constructing a path consisting only of edges due to Basic transition rules. By (a), it follows that this path will eventually reach a semi-terminal state. By (b), this semi-terminal state cannot be different from *FailState*, because $F \wedge G$ has no models. We derive that there is an edge leading to *FailState*. □

10.4.2 Proof of Lemma 8

For a state S in the graph $\text{GTL}^{\uparrow}_{F,G}$, we say that $S^{\downarrow}$ in $\text{GTL}_{F,G}$ is the *image* of S.

Lemma 8. *For any CNF formula F and a formula G formed from atoms occurring in F, if S' is a state reachable from $\emptyset||\emptyset$ in the graph $\text{GTL}_{F,G}$ then there is a state S in the graph $\text{GTL}^{\uparrow}_{F,G}$ such that $S^{\downarrow} = S'$.*

Proof. Since the property trivially holds for the initial state $\emptyset||\emptyset$, we only need to prove that all transition rules of $\text{GTL}_{F,G}$ preserve it.

Consider an edge $M||\Gamma \Longrightarrow M'||\Gamma'$ in the graph $\text{GTL}_{F,G}$ such that there is a state $M_1||\Gamma$ in the graph $\text{GTL}^{\uparrow}_{F,G}$ satisfying the condition $(M_1||\Gamma)^{\downarrow} = M||\Gamma$. We need to show that there is a state in the graph $\text{GTL}^{\uparrow}_{F,G}$ such that $M'||\Gamma'$ is its image in $\text{GTL}_{F,G}$. Consider several cases that correspond to a transition rule leading from $M||\Gamma$ to $M'||\Gamma'$:

Unit Propagate λ:

$$M||\Gamma \implies M\ l||\Gamma \ \text{ if } \ \begin{cases} C \vee l \in F \cup \Gamma \text{ and} \\ \overline{C} \subseteq M. \end{cases}$$

$M'||\Gamma'$ is $M\ l||\Gamma$. It is sufficient to prove that $M_1\ l^{C \vee l}||\Gamma$ is a state of $\text{GTL}^{\uparrow}_{F,G}$. It is enough to show that a clause $C \vee l$ is a reason for l to be in $M\ l$ with respect to $F \wedge G$, i.e., $F \wedge G \models C \vee l$ and $\overline{C} \subseteq M$. By the applicability conditions of *Unit Propagate* λ, $\overline{C} \subseteq M$. By the definition of a state $F \wedge G$ entails Γ. Since $C \vee l \in F \cap \Gamma$, $F \wedge G \models C \vee l$.

Test:

$$M\ ||\Gamma \Longrightarrow M\ \bar{l}||\Gamma \ \text{ if } \begin{cases} M \text{ is consistent,} \\ G \models \overline{M}, \\ l \in M. \end{cases}$$

$M'||\Gamma'$ is $M\ \overline{l}||\Gamma$. It is sufficient to prove that $M_1\ \overline{l}^{\overline{M}}||\Gamma$ is a state of $\text{GTL}^{\uparrow}_{F,G}$. $\overline{M}$ has the form $\overline{l} \vee C$. It is enough to show that a clause $\overline{l} \vee C$ is a reason for $\overline{l}$ to be in $M\ \overline{l}$ with respect to $F \wedge G$. It is trivial that $\overline{C} \subseteq M$. By applicability condition of the rule, $G \models \overline{l} \vee C$.

Backjump GT, *Decide*, *Fail*, and *Learn GT*: obvious. □

The process of turning a state of $\text{GTL}_{F,G}$ reachable from $\emptyset||\emptyset$ into a corresponding state of $\text{GTL}^{\uparrow}_{F,G}$ can be illustrated by the following example: Consider a formula F

$$\begin{array}{l} a \vee \neg b \\ \neg a \vee \neg b, \end{array} \tag{10.2}$$

a formula G

$$\neg b, \tag{10.3}$$

and a path in $\text{GTL}_{F,G}$

$$\begin{array}{l} \emptyset||\emptyset \implies (\mathit{Decide}) \\ b^{\Delta}||\emptyset \implies (\mathit{Unit\ Propagate}\ \lambda) \\ b^{\Delta}\ a||\emptyset \implies (\mathit{Test}) \\ b^{\Delta}\ a\ \neg a||\emptyset \end{array} \tag{10.4}$$

The construction in the proof of Lemma 8 applied to the nodes in this path gives following states of $\text{GTL}^{\uparrow}_{F,G}$:

$$\begin{array}{l} \emptyset||\emptyset \\ b^{\Delta}||\emptyset \\ b^{\Delta}\ a^{a\vee\neg b}||\emptyset \\ b^{\Delta}\ a^{a\vee\neg b}\ \neg a^{\neg a\vee\neg b}||\emptyset \end{array} \tag{10.5}$$

It is clear that these nodes form a path in $\text{GTL}^{\uparrow}_{F,G}$ with every edge justified by the same transition rule as the corresponding edge in path (10.4) in $\text{GTL}_{F,G}$.

10.4.3 Proof of Theorem 14$^{\uparrow}$

In this section F is an arbitrary and fixed CNF formula and G is an arbitrary and fixed formula formed from atoms occurring in F.

For a record M, by $lcp(M)$ we denote its largest consistent prefix. We say that a clause C is *conflicting* on a list M of literals if $F \wedge G$ entails C, and $\overline{C} \subseteq lcp(M)$.

For example, let M be the first component of the last state in (10.5):

$$b^{\Delta}\ a^{a \vee \neg b}\ \neg a^{\neg a \vee \neg b} \tag{10.6}$$

Then, $lcp(M)$ is obtained by dropping the last element $\neg a^{\neg a \vee \neg b}$ of M. It is clear that the reason $\neg a \vee \neg b$ for $\neg a$ to be in M is a conflicting clause on M.

Lemma 10. *The literal that immediately follows $lcp(M)$ in an inconsistent record M, has the form l^C where C is a conflicting clause on M.*

Proof. By the requirement (iii) of the definition of an extended record, the literal that immediately follows $lcp(M)$ may not be annotated by Δ. Consequently, the literal has the form l^C. We now show that C is a conflicting clause on M. Since C is a reason for l to be in $lcp(M)$ l^C, it immediately follows that $F \wedge G$ entails C, C can be written as $l \vee C'$, and $\overline{C'} \subseteq lcp(M)$. Since l immediately follows the largest consistent prefix of M, $\overline{l} \in lcp(M)$. Consequently, $\overline{C} \subseteq lcp(M)$. We derive that C is indeed a conflicting clause on M. □

For any inconsistent record $l_1 \cdots l_n$ and any conflicting clause C on this record, by $\beta_{l_1 \cdots l_n}(C)$ we denote the set of numbers i such that $l_i \in \overline{C}$. (It is clear that every element from $\overline{C}$ equals to one of the literals in $l_1 \cdots l_n$.) The relation $I < J$ between subsets I, J of $\{1 \cdots n\}$ is understood here as the lexicographical order between I and J sorted in descending order. For instance, $\{2\ 6\ 7\} < \{6\ 7\ 8\}$ because $7\ 6\ 2 < 8\ 7\ 6$ in lexicographical order.

Recall that the *resolution rule* can be applied to clauses $C \vee l$ and $C' \vee \neg l$ and produces the clause $C \vee C'$, called the *resolvent* of $C \vee l$ and $C' \vee \neg l$ on l.

Lemma 11. *Let M be a record and let l^B be a nondecision literal from $lcp(M)$. If clause D is the resolvent of B and a clause C conflicting on M then*

(i) D is a clause conflicting on M,

(ii) $\beta_M(D) < \beta_M(C)$.

For instance, let M be (10.6), let reason $a \vee \neg b$ for a in $lcp(M)$ be B, and let conflicting clause $\neg a \vee \neg b$ on M be C. Then D, the result of resolving B together

with C, is a clause $\neg b$. Lemma 11 asserts that $\neg b$ is a conflicting clause on M and that $\beta_M(D) < \beta_M(C)$. Indeed, $\beta_M(D) = \{1\}$ and $\beta_M(C) = \{2\ 1\}$.

Proof. (i) Clause D is a resolvent of B and C on some literal l'. Then, for some literal $l' \in B$, $\overline{l'} \in C$. The clause C can be written as $\overline{l'} \vee C'$.

In order to demonstrate that D is a conflicting clause we need to show that $\overline{D} \subseteq lcp(M)$ and $F \wedge G$ entails D.

Since B is a reason for l to be in $lcp(M)$, $F \wedge G$ entails B and B has the form $l \vee B'$ where $\overline{B'} \subseteq lcp(M)$. Since C is a conflicting clause on M, $\overline{C} \subseteq lcp(M)$ and $F \wedge G$ entails C. From the fact that $lcp(M)$ is consistent, it follows that there is no literal in B' such that its complement occurs in C. Consequently, $l' \notin B'$ so that l' is l and D is $B' \vee C'$. We derive that $\overline{D} \subseteq lcp(M)$. From the fact that $F \wedge G$ entails B, $F \wedge G$ entails C, and the construction of D, it follows that $F \wedge G$ entails D.

(ii) From the proof of (i) it follows that D is a resolvent of B and C on l where B has the form $l \vee B'$. Since B is a reason for l to be in $lcp(M)$, every literal in $\overline{B'}$ precedes l in $lcp(M)$. Since D is derived by replacing $\overline{l}$ in C with B', $\beta_M(D) < \beta_M(B)$. □

Let record M be $l_1 \cdots l_i \cdots l_n$, the *decision level* of a literal l_i is the number of decision literals in $l_1 \ldots l_i$: we denote it by $dec_M(l_i)$. We will also use this notation to denote the decision level of a set of literals: For a set $P \subseteq M$ of literals, $dec_M(P)$ is the decision level of the literal in P that occurs latest in M. For record M and a decision level j by M^j we denote the prefix of M that consists of the literals in M that belong to decision level less than j and by $M^{j]}$ we denote the prefix of M that consists of the literals in M that belong to decision level less or equal to j. For instance, let M be record (10.6) then $dec_M(a) = 1$, $dec_M(M) = 1$, M^1 is empty, and $M^{1]}$ is M itself.

Lemma 12. *For an inconsistent record M and a conflicting clause $l \vee C$ on M, if $dec_M(\overline{l}) > dec_M(\overline{c})$ for all $c \in C$ then $lcp(M)^{dec_M(\overline{C})]}\ l^{l \vee C}$ is a record.*

Proof. We need to show that (i) $l \notin lcp(M)^{dec(\overline{C})]}$ and (ii) $l \vee C$ is a reason for l to be in $lcp(M)^{dec(\overline{C})]}\ l$, i.e., $F \wedge G$ entails $l \vee C$ and $\overline{C} \subseteq lcp(M)^{dec(\overline{C})]}$.

Since $l \vee C$ is conflicting on M, $\overline{l \vee C} \subseteq lcp(M)$. From the consistency of $lcp(M)$ and the fact that $\overline{l} \in lcp(M)$, it follows that $l \notin lcp(M)$. Consequently, $l \notin lcp(M)^{dec(\overline{C})]}$.

Since $l \vee C$ is conflicting on M, $F \wedge G$ entails $l \vee C$ and $\overline{l \vee C} \subseteq lcp(M)$. Consequently, $\overline{C} \subseteq lcp(M)$. From the definition of $dec_M(\overline{C})$, it follows that $dec_M(\overline{C})$ is the decision level of the literal in $\overline{C}$ that occurs the latest in $lcp(M)$. By the definition of a decision level, $\overline{C} \subseteq lcp(M)^{dec_M(\overline{C})]}$. □

Theorem 14$^{\uparrow}$. *For any CNF formula F and a formula G formed from atoms occurring in F, the transition rule Backjump GT is applicable to any backjump state in* $\text{GTL}^{\uparrow}_{F,G}$.

Proof. Let $M||\Gamma$ be a backjump state in $\text{GTL}^{\uparrow}_{F,G}$. Let R be the list of reasons that are assigned to the nondecision literals in $lcp(M)$.

Consider the process of building a sequence $C_1, C_2, \ldots$ of clauses so that

- C_1 is the reason of the member of M that immediately follows $lcp(M)$, and
- C_j $(j > 1)$ is a resolvent of C_{j-1} and some clause in R

while derivation of new clauses is possible. From Lemma 11 (i) and the choice of C_1 and R, it follows that any clause in $C_1, C_2 \ldots$ is conflicting. By Lemma 11 (ii) we conclude that $\beta_M(C_j) < \beta_M(C_{j-1})$ $(j > 1)$. It is clear that this process will terminate after deriving some clause C_m, since the number of conflicting clauses on M is finite. It is clear that clause C_m cannot be resolved against any clause in R.

Case 1. C_m is the empty clause. Since $M||\Gamma$ is a backjump state, M contains a decision literal l^{Δ}. By part (iii) of the definition of a record, l belongs to $lcp(M)$. Consequently, M can be represented in the form $lcp(M)^{dec_M(l)}\ l^{\Delta}\ Q$.

By the choice of C_1, C_1 is a reason and must consist of at least one literal. Consequently, $m > 1$. Clause C_m is derived from clauses C_{m-1} and some clause in R. Since C_m is empty, C_{m-1} is a unit clause l'. We will show that

$$lcp(M)^{dec_M(l)}\ l^{\Delta}\ Q||\Gamma \Longrightarrow lcp(M)^{dec_M(l)}\ l'^{l'}||\Gamma$$

is an application of *Backjump GT*. It is sufficient to demonstrate that $lcp(M)^{dec_M(l)}\ l'^{l'}$ is a record. Since $lcp(M)^{dec_M(l)}\ l^{\Delta}\ Q$ is a record, we only need to show that $l' \notin lcp(M)^{dec_M(l)}$ and clause l' is a reason for l' to be in $lcp(M)^{dec_M(l)}\ l'$. Recall that C_{m-1}, i.e., l', is a conflicting clause. Consequently, $F \wedge G$ entails l' and $\overline{l'} \in lcp(M)$. Since $lcp(M)$ is consistent, $l' \notin lcp(M)$ so that $l' \notin lcp(M)^{dec_M(l)}$.

On the other hand, from the fact that $F \wedge G$ entails l' it immediately follows that clause l' is a reason for l' to be in $lcp(M)^{dec_M(l)}\ l'$

Case 2. C_m is not empty. Since C_m is a conflicting clause on M, the complement of any literal in C_m belongs to $lcp(M)$. Furthermore, every such complement is a decision literal in $lcp(M)$. Indeed, if this complement is $\overline{l}^{\overline{l} \vee B} \in lcp(M)$ then $\overline{l} \vee B$ is one of the clauses B_i, and it can be resolved against C_m.

By the definition of a decision level, there is at most one decision literal that belongs to any decision level. It follows that C_m can be written as $l \vee C'_m$ so that $dec_M(\overline{l}) > dec_M(\overline{c})$ for any $c \in C'_m$. Consequently, M can be written as $lcp(M)^{dec_M(\overline{l})}\ \overline{l}^{\Delta}\ Q$. Note that

$$lcp(M)^{dec_M(\overline{l})}\ \overline{l}^{\Delta}\ Q||\Gamma \Longrightarrow lcp(M)^{dec_M(\overline{C'_m})]}\ l^{C_m}||\Gamma$$

is an application of *Backjump GT*. Indeed, by Lemma 12 $lcp(M)^{dec_M(\overline{C}'_m)]}\ l^{C_m}$ is a record. □

For a CNF formula F and a formula G formed from atoms occurring in F, we say that a record M is a *backjump record* with respect to $F \wedge G$ if $M||\Gamma$ is a backjump state in $\text{GTL}^{\uparrow}_{F,G}$. Algorithm 3 presents procedure *BackjumpClause* that computes a backjump clause for any backjump record with respect to a formula.

```
BackjumpClause(M)
Arguments      : M is a backjump record with respect to F ∧ G
Return Value   : C is a backjump clause
begin
    C ← the reason of the member of M that immediately follows lcp(M)
    N ← the list of the nondecision literals in lcp(M)
    R ← the list of the reasons that are assigned to the literals in N
    while C̄ ∩ N ≠ ∅ do
        l ← a literal in C̄ ∩ N
        B ← the clause in R that contains l
        C' ← the resolvent of C and B on l
        if C' = ∅ then
            return C
        C ← C'
    return C
end
```

Algorithm 3: A procedure for generating a backjump clause.

$lcp(M)$	$b^{\Delta}\ a^{a\vee\neg b}$
C_1	$\neg a \vee \neg b$
N	$a^{a\vee\neg b}$
R	$a \vee \neg b$
C_2	$\neg b$ is the resolvent of C_1 and $a \vee \neg b$

Figure 10.1: Sample execution of the *BackjumpClause* algorithm on backjump record (10.6) with respect to $F \wedge G$ where F is (10.2) and G is (10.3).

The algorithm follows from the construction of the proof of Theorem $14^{\uparrow}$. It is based on the iterative application of the resolution rule on reasons of the smallest inconsistent prefix of a state.

The proof of Theorem $14^{\uparrow}$ allows us to conclude the termination of *BackjumpClause* and asserts that a clause returned by the procedure is a backjump clause on a backjump state.

For instance, let F be (10.2) and G be (10.3). Consider an execution of *BackjumpClause* on backjump record (10.6) with respect to $F \wedge G$. Figure 10.1 illustrates what the values of $lcp(M)$, C, N, and R are during the execution of the *BackjumpClause* algorithm. By C_i we denote a value of C before the i-th iteration of the **while** loop.

The algorithm will terminate with the clause $\neg b$. The proof of Theorem $14^{\uparrow}$ asserts that (i) this clause is a backjump clause such that b is a decision literal in M and (ii) the transition

$$b^{\Delta}\ a^{a\vee\neg b}\ \neg a^{\neg a\vee\neg b}||\emptyset \Longrightarrow$$
$$\neg b^{\neg b}||\emptyset$$

in $\text{GTL}^{\uparrow}{}_{F,G}$ is an application of *Backjump GT*. Indeed, by Lemma 12 $\neg b^{\neg b}$ is a record.

Note that a backjump clause may be derived in other ways than captured by the *BackjumpClause* algorithm: the transition rule *Backjump GT* is applicable with an arbitrary backjump clause. Usually, DPLL-like procedures implement conflict-driven backjumping and learning where a particular learning schema such as, for instance, *Decision* and *FirstUIP* [Mitchell, 2005] is applied for computing a special kind of a backjump clause. It turns out that the *BackjumpClause* algorithm captures the *Decision* learning schema for the generate and test algorithm. Typically, SAT solvers impose an order for resolving the literals during the process of *Decision*

backjump clause derivation. We can impose similar order by replacing the line

$$l \leftarrow \text{a literal in } \overline{C} \cap N$$

in the algorithm *BackjumpClause* with

$$l \leftarrow \text{a literal in } \overline{C} \cap N \text{ that occurs latest in } lcp(M).$$

This section introduced the *BackjumpClause* algorithm that derives a *Decision* backjump clause for an arbitrary backjump state. In the next section we will introduce an algorithm that will compute a generate and test counterpart of *FirstUIP* backjump clause.

10.5 Generate and Test: FirstUIP Conflict-Driven Backjumping and Learning

Conflict-driven backjumping and learning proved to be a highly successful technique in modern SAT solving. Furthermore, in [Zhang *et al.*, 2001] the authors investigated the performance of various learning schemes and established experimentally that the *FirstUIP* clause is the most useful single clause to learn. In this section we will present an algorithm for computing the *FirstUIP* clause for the generate and test algorithm.

There are two common methods for describing a backjump clause construction in SAT literature. The first one employes the implication graph [Marques-Silva and Sakallah, 1996a]. The second method used in SAT for characterizing a backjump clause derivation employes resolution. In the previous section we used the graph $\text{GTL}^{\uparrow}_{F,G}$ to describe the *BackjumpClause* algorithm for computing a *Decision* backjump clause that follows the second tradition. This section presents the *BackjumpClauseFirstUIP* algorithm for computing a generate and test counterpart of a *FirstUIP* backjump clause by means of $\text{GTL}^{\uparrow}_{F,G}$ and resolution.

Algorithm 4 presents the procedure *BackjumpClauseFirstUIP* that computes the *FirstUIP* backjump clause for any backjump record with respect to a formula $F \wedge G$.

We now state the correctness of the algorithm *BackjumpClauseFirstUIP*. We start by showing its termination. By C_1 we denote the initial value assigned to clause C. From Lemma 11 (i) and the choice of C_1 we conclude that at any

```
BackjumpClauseFirstUIP(M)
Arguments      : M is a backjump record with respect to F ∧ G
Return Value   : C is a backjump clause
begin
    C ← the reason of the member of M that immediately follows lcp(M)
    l ← the literal in ¬C that occurs latest in lcp(M)
    P ← the sublist of lcp(M) that consists of the literals that belong to the
    decision level dec(l)
    R ← the list of the reasons that are assigned to the literals in P
    while |¬C ∩ P| > 1 do
        l ← the literal in ¬C that occurs latest in P
        B ← the clause in R that contains l
        C ← the resolvent of C and B on l
    return C
end
```

Algorithm 4: A procedure for generating a FirstUIP backjump clause.

point of computation clause C is conflicting on M. By Lemma 11 (ii), the value of $\beta_M(C)$ decreases with each new assignment of clause C in the **while** loop. It follows that the **while** loop will terminate since the number of conflicting clauses on M such that $|\overline{C} \cap P| > 1$ is finite. By C_m we will denote the clause C with which the **while** loop terminates. In other words *BackjumpClauseFirstUIP* returns C_m. We now show that C_m is indeed a backjump clause. We already concluded that C_m is a conflicting clause on M. Furthermore, from the termination condition of the **while** loop $|\overline{C_m} \cap P| \leq 1$. From the choice of C_1 and P it follows that $|\overline{C_m} \cap P| = 1$. Consequently, C_m can be written as $l \vee C'_m$ where $\overline{l}$ is in singleton $\overline{C_m} \cap P$. By Lemma 11 (ii), $\beta(C_m) \leq \beta(C_1)$. From the definition of β and the choice of P it follows that $dec_M(\overline{l}) > dec_M(\overline{c})$ for all $c \in C'_m$. By Lemma 12, $lcp(M)^{dec_M(\overline{C'_m})]}\ l^{C_m}$ is a record. In other words, transition

$$M||\Gamma \Longrightarrow lcp(M)^{dec_M(\overline{C'_m})]}\ l^{C_m}||\Gamma$$

is an application of *Backjump GT*. Consequently, C_m is a backjump clause.

Chapter 11

Extending Cmodels Algorithm to Nontight Programs by Means of "Generate and Test" DPLL

Some of the methods presented in this chapter are parts of joint work with Enrico Giunchiglia and Marco Maratea [Giunchiglia *et al.*, 2004a; 2006]. The main topic of this chapter is the ASP-SAT *with Learning* algorithm for computing answer sets of an arbitrary, possibly nontight, program. The procedure ASP-SAT *with Learning*, unlike ASSAT (Section 9.2), modifies the SAT solver algorithm. This allows us

- to incorporate testing of a computed model inside the solver (and hence to avoid multiple invocations of a SAT solver),
- to utilize conflict-driven learning do guide a search of a SAT solver by means of loop formulas by learning clauses based on loop formulas on demand.

Before introducing ASP-SAT *with Learning* we present a simpler algorithm ASP-SAT [Giunchiglia *et al.*, 2006] in Section 11.1. We use the graph $\mathrm{GT}_{F,G}$ to describe the ASP-SAT procedure. Section 11.2 presents ASP-SAT *with Learning*. Section 11.3 describes the theory used in the CMODELS implementation of the application of the transition rule *Test* of the ASP-SAT *with Learning* algorithm discussed in Section 11.4. Section 11.5 provides details on the implementation of a variant of ASP-SAT *with Learning* in CMODELS by means of incremental SAT-solving. In this section

we also discuss the advantage of the ASP-SAT *with Learning* method implemented in CMODELS over the ASSAT procedure. In Section 11.6 we present an experimental analysis. Section 11.7 briefly describes the results of First and Second Answer Set Programming System Competitions.

11.1 ASP-SAT Algorithm

In Section 5.3 we demonstrated a method for specifying the algorithm of an answer set solver by means of the graph SM_Π. We use this method to describe the ASP-SAT algorithm [Giunchiglia *et al.*, 2006] using the graph $\text{GT}_{F,G}$. The application of the ASP-SAT algorithm to a program Π can be viewed as constructing a path from $\emptyset$ to a terminal node in the graph $\text{GT}_{ED\text{-}Comp(\Pi),LF(\Pi)}$ where ASP-SAT assigns priorities to the inference rules of $\text{GT}_{ED\text{-}Comp(\Pi),LF(\Pi)}$ as follows:

Backtrack, *Fail* >>
Unit Propagate >>
Decide >>
Test.

In comparison with the procedure of ASSAT (Section 9.2), the advantage of ASP-SAT is that it is guaranteed to work in polynomial space. On the other hand, there may be exponentially more models of the program's completion than answer sets. Unlike ASSAT that prunes the search space of a SAT solver by introducing loop formulas of the program that may eliminate unwanted models, ASP-SAT will enumerate all the models of the program. This is the main drawback of the ASP-SAT procedure. Section 11.2 describes the ASP-SAT *with Learning* algorithm that implements a procedure similar to ASP-SAT with pruning the search space in a manner of ASSAT. As a step in this direction in Section 10.2 we will extend the generate and test graph introduced in this section with backjumping and learning.

11.2 ASP-SAT with Learning: Cmodels Algorithm for Nontight Programs

In Section 11.1 we described the ASP-SAT algorithm that utilizes the generate and test approach for computing answer sets of a program. In [Giunchiglia *et al.*, 2006] a more general algorithm ASP-SAT *with Learning* was introduced that took advantage of such sophisticated techniques as learning and backjumping. Here we use the graph $\text{GTL}^{\uparrow}_{F,G}$ for specifying the ASP-SAT *with Learning* algorithm, similarly as we used $\text{GT}_{F,G}$ for specifying the ASP-SAT algorithm in Section 11.1. In particular we will assign priorities to the transition rules of $\text{GTL}^{\uparrow}_{F,G}$.

The application of the ASP-SAT *with Learning* algorithm to a program Π can be viewed as constructing a path from $\emptyset||\emptyset$ to a terminal node in the graph $\text{GTL}^{\uparrow}_{F,G}$, where

- F is *ED-Comp*(Π) – the completion *Comp*(Π) converted to CNF using the ED-transformation, and
- G is $LF(\Pi)$.

The ASP-SAT *with Learning* algorithm assigns priorities to the inference rules of $\text{GTL}^{\uparrow}_{F,G}$ as follows:

Backjump GT, Fail >>
Unit Propagate λ >>
Decide >>
Test.

Also, ASP-SAT *with Learning* always applies the transition rule *Learn GT* in a non-semi-terminal state reached by an application of *Backjump GT*.

The priorities imposed on the rules by ASP-SAT *with Learning* guarantee that the transition rule *Test* is applied only in states $M||F, \Gamma$ where M is a model of $F \cup \Gamma$ (clausified completion F extended by learned clauses Γ). This allows ASP-SAT *with Learning* to proceed with its search in case if the model is not an answer set. Furthermore, the ASP-SAT *with Learning* strategy guarantees that in a state reached by an application of *Test*, first *Backjump GT* will be applied and then in the resulting state *Learn GT* will be applied. In Section 11.4 we describe the methods used by CMODELS (i) to decide whether *Test* is applicable to a state $M||\Gamma$ where M

is a model of Γ and (ii) to construct clauses learned due to such applications of *Learn* GT. These clauses are constructed using loop formulas. In this sense ASP-SAT *with Learning*, similarly to the ASSAT algorithm, uses loop formulas to guide its search.

In Section 3.4 we demonstrated how the graph DPL_F may be extended with the transition rules *Restart* and *Forget* that describe such techniques as restarts and forgetting. By adding these rules to the definition of the graph $\text{GTL}^{\uparrow}_{F,G}$ we allow the ASP-SAT *with Learning* algorithm to utilize corresponding techniques. In fact, by incorporating forgetting into ASP-SAT *with Learning* we permit the algorithm to avoid exponential blow-up in space by periodically discarding previously learned clauses so that it is guaranteed to work in polynomial space.

The system CMODELS implements an algorithm ASP-SAT *with Learning*. The correctness of the CMODELS algorithm immediately follows from Theorem $13^{\uparrow}$.

11.3 Terminating Loops

In [Lin and Zhao, 2002], the authors defined the notion of a terminating loop for traditional programs. In case when a computed model of completion is not an answer set of a program, such loops can be used for finding a loop formula unsatisfied by the model. Here we extend the notion of a terminating loop to semi-traditional programs and state how such a loop can be computed.

Given a set X of atoms, by $G_{\Pi,X}$ we denote a subgraph of the dependency graph (Sections 4.5, 7.6) of a program Π induced by X.

Recall that we identify a set of atoms with the truth assignment that maps the elements of the set to *True*, and all other atoms to *False*.

Let Π be a semi-traditional program.

A loop L is *terminating* under set Y of atoms if L is a strongly connected component of $G_{\Pi,Y}$ and there does not exist another strongly connected component L' of $G_{\Pi,Y}$ such that there is a path from any vertex in L to a vertex in L'.

The following theorem states that given a program Π and a set X of atoms such that X is a model of $Comp(\Pi)$ and X is not an answer set of Π, to calculate a loop formula of Π unsatisfied by X it is sufficient to

- find any proper subset X' of X such that $X' \models \Pi^X$ (It is easy to see that the minimal set of atoms satisfying Π^X is a proper subset of X. For semi-

traditional programs calculating the minimal set of atoms satisfying Π^X can be done in linear time),

- compute a loop formula of any terminating loop in $G_{\Pi,X\setminus X'}$.

Theorem 15. *For a semi-traditional program* Π*, and sets* X*,* X' *of atoms such that* X *is a model of* $Comp(\Pi)$*,* $X' \subset X$*, and* X' *is a model of* Π^X*, there is a terminating loop of* Π *under* $X \setminus X'$*. Furthermore,* X *does not satisfy the loop formula of any terminating loop of* Π *under* $X \setminus X'$*.*

We provide a proof of this theorem in Section 13.6.

11.4 Cmodels Algorithm: *Test* Application

Recall that for a set M of literals by M_Π we denote all literals in M whose atoms occur in Π.

There is an efficient procedure for deciding whether the transition rule *Test* is applicable to a state $M||\Gamma$ in the graph $\text{GTL}^{\uparrow}_{ED\text{-}Comp(\Pi),LF(\Pi)}$ where M is a model of *ED-Comp*(Π). Indeed, *Test* is applicable to $M||\Gamma$ if and only if M^+_Π does not correspond to any answer set of a program Π. For semi-traditional programs verifying whether M^+_Π is a minimal set of atoms satisfying $\Pi^{M^+_\Pi}$ can be done in linear time. This fact permits an efficient execution of CMODELS whose strategy is to apply *Test* only to such states.

The proof of Theorem $13^{\uparrow}$ provides a way of constructing a reason C for a literal l given a state $M||\Gamma$ such that there is the transition $M||\Gamma \implies M\ l^C||\Gamma$ due to *Test*. This reason C is $\overline{M}$. In practice, the shorter the reason C is the more information it provides to a SAT solver for the future search. The system CMODELS implements two different methods for computing short reasons. Both of the methods are based on loop formulas. We start by describing an "atomreason" method and continue with a "loopformulareason" method.

Let $M||\Gamma$ be a state in $\text{GTL}^{\uparrow}_{ED\text{-}Comp(\Pi),LF(\Pi)}$ such that M is a model of *ED-Comp*(Π) and *Test* is applicable to $M||\Gamma$. The atomreason approach requires a loop L such that for its loop formula F_L, $M \not\models F_L$. In Section 11.3 we discussed how such a loop can be computed. Recall that a loop formula F_L has the form (9.3)

(Section 9.1). This loop formula can be easily rewritten in disjunctive normal form:

$$\bigwedge_{l \in L} \overline{l} \vee \bigvee R(L) \tag{11.1}$$

Since $M \not\models F_L$ it follows that for any $R(L)$ there is a literal l' in $R(L)$ so that $\overline{l}' \in M$. Similarly there is l' in $\bigwedge_{l \in L} \overline{l}$ so that $\overline{l}' \in M$. We can construct a clause C that consists of such literals l' taken from all conjunctions of F_L. It is easy to see that for any literal $c \in C$, C is a reason for c to be in M c with respect to *ED-Comp*$(\Pi) \wedge LF(\Pi)$. The system CMODELS with the option flag `-atomreason` will only consider edges due to the transition rule *Test* of the kind $M||\Gamma \implies M\ c^C||\Gamma$ where C is computed by the described procedure. This concludes the description of the atomreason method.

Let $M||\Gamma$ be a state in $\text{GTL}^{\uparrow}_{\textit{ED-Comp}(\Pi),LF(\Pi)}$ such that M is a model of *ED-Comp*(Π) and *Test* is applicable to $M||\Gamma$. The loopformulareason method, as the atomreason method, starts by finding a loop formula F_L of the form (11.1). Recall that every conjunction $R(L)$ stands for a body of some rule in Π whose head is an atom. It follows that for any $R(L)$ containing more than one atom, there is a corresponding explicit definition atom r_l in *ED-Comp*(Π) (see Section 6.3) that denotes $R(L)$. We can construct a clause C from F_L by replacing

- $\bigwedge_{l \in L} \overline{l}$ with l' so that $\overline{l}' \in M$ (since $M \not\models F_L$ it follows that l' exists),
- each of its conjunctions $R(L)$ that contain more than one atom with corresponding r_l.

It is easy to see that for any literal $c \in C$, C is a reason for c to be in M c with respect to *ED-Comp*$(\Pi) \wedge LF(\Pi)$. The system CMODELS with the default settings will only consider edges due to the transition rule *Test* of the kind $M||\Gamma \implies M\ c^C||\Gamma$ where C is computed by the loopformulareason procedure. This concludes the description of the loopformulareason method.

We note that CMODELS implementing the atomreason method corresponds to the ASP-SAT *with Learning* algorithm introduced in [Giunchiglia *et al.*, 2004a; 2006].

11.5 Incremental SAT-solving for SAT-based ASP

Incremental SAT-solving allows the user to solve several SAT problems $\Gamma_1, \ldots, \Gamma_n$ one after another, if Γ_{i+1} results from Γ_i by adding clauses, so that the solution to Γ_{i+1} may benefit from the knowledge obtained during solving $\Gamma_1, \ldots, \Gamma_i$.

Nowadays, various SAT-solvers that implement interfaces for incremental SAT solving are available, as for instance ZCHAFF and MINISAT.

We can use incremental SAT solving as means for implementing CMODELS in the following way. Given a program Π:

1. Take Γ to be *ED-Comp*(Π),
2. Call an incremental SAT solver with the set Γ;
   (a) If it returns no model, exit with no answer set; otherwise
   (b) verify whether the model is an answer set. If so, output the answer set and exit. If not, extend Γ by a clause computed by the atomreason or loopformulareason methods and proceed to step 2.

For simplicity of presentation, this procedure describes the mechanism for finding a single answer set. We call this approach CMODELS+.

Using incremental SAT solvers for implementing CMODELS is related to the ASSAT system approach.

1. Like ASSAT, CMODELS+ requires only programming the interface to a SAT solver that permits transforming the data to and from the solver.
2. Within CMODELS+ it is essential for its termination that learned clauses from verification step 2b are never deleted. Therefore, unlike CMODELS that is guaranteed to work in polynomial space, CMODELS+ may require exponential space, like ASSAT.

Nevertheless, it has an advantage over the ASSAT procedure. In case when a SAT solver is invoked repeatedly, CMODELS+ permits reusing some information, which the SAT solver recalls from previous invocations. This allows the solver to disregard some parts of the search tree. On the other hand, CMODELS+ is not as good in this sense as CMODELS, which never explores the same parts of a search tree twice.

11.6 Experimental Analysis

As in Section 6.5.2 we describe here experiments that were conducted using the system whose technical specifications are presented in Section 6.5. In this section, we also compare the performance of CMODELS with ASSAT version 2.02 (Section 9.2), SMODELS and SMODELS$_{cc}$. Details on the versions of answer set solvers CMODELS, SMODELS, and SMODELS$_{cc}$ are provided in Section 6.5. As before, our experiments compare the systems performance on the problem of finding only one answer set. In all experiments reported here,

- system CMODELS using ZCHAFF implements the CMODELS+ algorithm,
- system CMODELS using MINISAT implements the CMODELS+ algorithm,
- system ASSAT uses the SAT solver CHAFF.

The nontight benchmarks that we used are *Deterministic Automaton*, *Random Nontight*, *Wire Routing*, *Bounded Model Checking*, *Hamiltonian Cycle*. Here are their brief descriptions:

- The *Deterministic Automaton* is the problem of checking requirements in a deterministic automaton and are described in [Ştefănescu *et al.*, 2003]
- *Random Nontight* is a collection of randomly generated nontight programs.
- A *Wire-Routing* in a n x n grid consists of finding routes for w wires given their start and end points so that the following constraints are satisfied:
  1. No grid edge is used by more than one wire,
  2. Some grid points allow for two wires to cross; all other points can be used by at most one wire,
  3. Some areas of the grid are blocked and cannot be "touched" by any wire.
- *Bounded Model Checking* is a the task of verification of asynchronous concurrent systems described in [Heljanko and Niemelä, 2003].
- A Hamiltonian cycle in an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is a cycle in G such that every vertex in V occurs exactly once in the cycle. The input of the *Hamiltonian Cycle* problem is an undirected graph, and the goal is to find a Hamiltonian cycle in it.

Instance	CMODELS		ASSAT	SMODELS	SMODELS$_{cc}$
	MINISAT	ZCHAFF			
detA.IDFD.mutex3	0.02	0.02	0.21	0.04	0.12
detA.IDFD.mutex4	0.48	0.68	0.22	17.96	52.71
detA.IDFD.phi4	0.03	0.02	0.66	0.05	0.17
detA.IDFD.phi5	0.18	0.09	21.11	1.0	3.88
detA.Morin.mutex3	0.02	0.03	0.03	0.05	0.16
detA.Morin.mutex4	0.76	1.31	1.19	21.46	64.81
detA.Morin.phi4	0.04	0.04	0.71	0.07	0.22
detA.Morin.phi5	0.23	0.18	26.97	1.17	4.43
random.n40-sat-b10	0.17	0.62	0.2	1.06	6.12
random.n40-sat-b11	0.06	2.05	0.53	0.35	10.27
random.n40-sat-b12	0.03	0.05	0.71	0.6	1.16
random.n40-sat-b1	0.03	3.56	1.56	3.45	8.56
random.n40-sat-b2	0.04	0.88	0.12	0.67	1.18
random.n50-sat-b10	2.85	0.31	122.37	12.14	121.63
random.n50-sat-b11	0.72	2.86	8.6	42.6	560.22
random.n50-sat-b12	3.52	37.78	9.02	7.89	t-o
random.n50-sat-b1	2.29	0.22	16.56	1.83	13.67
random.n50-sat-b2	0.45	62.92	19.11	45.95	5.64
random.n60-sat-b10	4.57	t-o	t-o	282.79	162.4
random.n60-sat-b11	0.79	t-o	142.25	389.25	t-o
random.n60-sat-b12	78.48	128.04	535.81	8.31	30.33
random.n60-sat-b1	2.76	t-o	t-o	t-o	t-o
random.n60-sat-b2	16.81	90.28	t-o	t-o	t-o
random.n40-unsat-b10	0.16	0.8	0.43	1.1	4.97
random.n40-unsat-b11	0.26	0.93	0.93	2.66	11.59
random.n40-unsat-b12	0.2	1.14	0.51	1.51	7.9
random.n40-unsat-b1	0.22	1.23	0.76	1.63	10.03
random.n40-unsat-b2	0.16	0.84	0.59	1.42	6.97
random.n50-unsat-b10	12.82	135.08	170.22	88.53	t-o
random.n50-unsat-b11	7.0	81.84	81.15	40.17	552.31
random.n50-unsat-b12	3.98	33.5	75.07	57.92	t-o
random.n50-unsat-b1	2.1	20.09	14.06	30.76	241.72
random.n50-unsat-b2	14.52	139.23	354.66	80.84	t-o

Figure 11.1: Deterministic Automaton, Random; runtimes of CMODELS using MINISAT, CMODELS using ZCHAFF, ASSAT, SMODELS, SMODELS$_{cc}$.

Instance	CMODELS		SMODELS	SMODELS$_{cc}$
	MINISAT	ZCHAFF		
wire.10.x.10.b.5.a.25S	0.5	1.41	t-o	134.11
wire.10.x.10.b.5.a.35S	0.26	1.41	13.37	7.3
wire.10.x.10.b.5.a.20U	2.33	50.72	41.17	4.84
wire.12.x.12.b.5.a.15U	t-o	t-o	t-o	t-o
wire.12.x.12.b.5.a.20U	349.83	t-o	t-o	t-o
dp-10.fsa-D-i-O2-b10	0.11	0.1	119.14	6.43
dp-12.fsa-D-i-O2-b9	159.57	144.65	454.57	t-o
dp-6.fsa-D-i-O2-b6	0.02	0.02	0.04	0.19
dp-8.fsa-D-i-O2-b8	0.05	0.03	1.51	0.72
hc-1S	0.55	1.02	t-o	36.35
hc-2S	2.29	10.83	t-o	16.09
hc-3S	8.98	1.4	t-o	t-o
hc-4S	1.66	4.36	0.82	3.78
hc-5U	0.02	0.01	0.02	0.03
hc-6U	t-o	t-o	t-o	t-o
hc-7U	0.02	0.01	0.02	0.03
hc-8U	0.01	0.02	0.02	0.03

Figure 11.2: Wire Routing, Bounded Model Checking, Hamiltonian Cycle; runtimes of CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, SMODELS$_{cc}$.

Figure 11.1 reports the performance of the five systems CMODELS using MINISAT, CMODELS using ZCHAFF, ASSAT, SMODELS, SMODELS$_{cc}$ on Deterministic Automaton and Random problems. These problems are encoded using traditional rules. The system ASSAT only supports traditional programs.

Figure 11.2 reports the performance of the answer set solvers CMODELS using MINISAT, CMODELS using ZCHAFF, SMODELS, and SMODELS$_{cc}$ on Wire Routing, Bounded Model Checking, Hamiltonian Cycle problems. These problems are encoded using choice and cardinality constraints rules therefore we do not run ASSAT on these instances.

We do not report grounding times in Figures 11.1 and 11.2 because all answer set solvers used the same grounded instances.

It is easy to see that CMODELS using MINISAT or ZCHAFF often outperforms other systems.

11.7 First and Second Answer Set Programming System Competitions

There were two answer set programming system competitions held in recent years.

The First Answer Set Programming System Competition[1] [Gebser *et al.*, 2007c] was held in conjunction with the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07) in 2007. There were three tracks and ten systems in the competition for nondisjunctive programs. The System CMODELS took the third place in SCore (Solver, Core Language) track. The Systems CLASP[2] and SMODELS took the first and the second place respectively in this track. In Section 14.1 we describe how the answer set solver CLASP can be seen as an enhancement of the approach pioneered by CMODELS.

The Second Answer Set Programming System Competition[3] [Denecker *et al.*, 2009] was held in conjunction with the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09) in 2009. There were sixteen participating solvers and two tracks: Decision Problems and Optimization Problems. CMODELS took part in the first track. It took the second place as a single system team. The Systems CLASPFOLIO[4] and DLV took the first and the third places respectively. It is interesting to note that CLASPFOLIO is a portfolio solver based on CLASP: it is a python-script which predicts the best options for CLASP to solve a problem in question.

[1] http://asparagus.cs.uni-potsdam.de/contest/index.php .
[2] http://potassco.sourceforge.net/ .
[3] http://www.cs.kuleuven.be/∼dtai/events/ASP-competition/index.shtml .
[4] http://www.cs.kuleuven.be/ dtai/events/ASP-competition/Teams/Claspfolio.shtml .

Chapter 12

Description of Abstract Answer Set Solvers with Learning

In this chapter we will extend the graph SM_Π introduced in Section 5.2 to capture backjumping and learning for the abstract SMODELS algorithm in a similar manner as we extended the graph $\text{GT}_{F,G}$ to $\text{GTL}_{F,G}$ in Section 10.2 to capture backjumping and learning for the generate and test algorithm. As a result we will be able to model the algorithms of such answer set solvers as SUP [Lierler, 2008] and SMODELS_{cc} [Ward and Schlipf, 2004]. We note that the development of the system SUP was inspired by the work on the abstract framework for describing answer set solvers presented in this dissertation. In Section 12.1 we present the graph SML_Π that extends the graph SM_Π with backjumping and learning. Section 12.2 introduces the graph $\text{SML}_\Pi^\uparrow$ similar as in Section 10.3 we introduced $\text{GTL}_{F,G}^\uparrow$ for $\text{GTL}_{F,G}$. Section 12.3 presents the proofs for the theoretical findings discussed in this chapter. In Section 12.4 we use the graph $\text{SML}_\Pi^\uparrow$ to describe the *BackjumpClause* and *BackjumpClauseFirstUIP* algorithms for computing the *Decision* and *FirstUIP* backjump clauses. Section 12.5 first presents the SUP algorithm, second describes the algorithm underlying answer set solver SMODELS_{cc}, and third illustrates their differences. Section 12.6 provides implementation details of the system SUP and reports experimental analysis on its performance.

Backchain False λ:

$$M||\Gamma \implies M\ \overline{l}||\Gamma \text{ if } \begin{cases} a \leftarrow l, B \in \Pi \cup \Gamma, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M \end{cases}$$

Backjump LP:

$$P\ l^{\Delta}\ Q||\Gamma \implies P\ l'||\Gamma \text{ if } \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent and} \\ \Pi \text{ entails } l' \vee \overline{P} \end{cases}$$

Learn LP:

$$M||\Gamma \implies M||\ \leftarrow B,\ \Gamma \text{ if } \Pi \text{ entails } \overline{B}$$

Figure 12.1: The additional transition rules of the graph SML_Π.

12.1 Graph SML_Π

An *(augmented) state* relative to a program Π is either a distinguished state *FailState* or a pair of the form $M||\Gamma$ where M is a record relative to the set of atoms occurring in Π, and Γ is a (multi)set of constraints formed from atoms occurring in Π that are entailed by Π.

For any program Π, we will define a graph SML_Π. Its nodes are the augmented states relative to Π. The transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Unfounded*, *Decide* and *Fail* of SM_Π are extended to SML_Π as follows: $M||\Gamma \implies M'||\Gamma$ ($M||\Gamma \implies$ *FailState*) is an edge in SML_Π justified by a transition rule T if and only if $M \implies M'$ ($M \implies$ *FailState*) is an edge in SM_Π justified by T. Figure 12.1 presents the other transition rules of SML_Π.

We refer to the transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Backchain False* λ, *Unfounded*, *Backjump LP*, *Decide*, and *Fail* of the graph SML_Π as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn LP* is applicable to it.

The graph SML_Π can be used for deciding whether a program Π has an answer set by constructing a path from $\emptyset||\emptyset$ to a semi-terminal node:

Theorem 16. *For any program* Π*,*

(a) every path in SML_Π *contains only finitely many edges labeled by Basic transition rules,*

(b) for any semi-terminal state $M||\Gamma$ *of* SML_Π *reachable from* $\emptyset||\emptyset$, M^+ *is an answer set of* Π,

(c) FailState is reachable from $\emptyset||\emptyset$ *in* SML_Π *if and only if* Π *has no answer sets.*

Thus if we construct a path from $\emptyset||\emptyset$ so that Basic transition rules periodically appear in it then some semi-terminal state will be eventually reached; as soon as a semi-terminal state is reached the problem of finding an answer set is solved.

For instance, let Π be program (4.8). Here is a path in SML_Π:

$$\begin{array}{ll}
\emptyset||\emptyset & \Longrightarrow \ (\textit{Decide}) \\
a^\Delta||\emptyset & \Longrightarrow \ (\textit{Unit Propagate LP}) \\
a^\Delta\ c||\emptyset & \Longrightarrow \ (\textit{All Rules Cancelled}) \\
a^\Delta\ c\ \neg b||\emptyset & \Longrightarrow \ (\textit{Decide}) \\
a^\Delta\ c\ \neg b\ d^\Delta||\emptyset & \Longrightarrow \ (\textit{Unfounded}) \\
a^\Delta\ c\ \neg b\ d^\Delta\ \neg d||\emptyset & \Longrightarrow \ (\textit{Backjump LP}) \\
a^\Delta\ c\ \neg b\ \neg d||\emptyset & \Longrightarrow \ (\textit{Learn LP}) \\
a^\Delta\ c\ \neg b\ \neg d||\neg a \vee \neg c \vee b \vee \neg d &
\end{array} \tag{12.1}$$

Since the state $a^\Delta\ c\ \neg b\ \neg d$ is semi-terminal, Theorem 16 (b) asserts that

$$\{a, c, \neg b, \neg d\}^+ = \{a, c\}$$

is an answer set for Π.

The proof of Theorem 16 can be found in Section 12.3.

As in case of the graphs DP_F and DPL_F, *Backjump LP* is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$ (Theorem 17, Section 12.2), and is essentially a generalization of the transition rule *Backtrack* of the graph SM_Π.

In Section 3.4 we demonstrated how the graph DPL_F may be extended with the transition rules *Restart* and *Forget* that characterize the restarts and forgetting techniques (Section 3.3) commonly implemented in modern SAT solvers. Similarly, we may extend the graph SML_Π with the transition rule *Restart* of DPL_F and the

following transition rule

$$\begin{array}{l}\textit{Forget LP}:\\ M||\leftarrow B,\ \Gamma \implies M||\Gamma.\end{array}$$

to capture the ideas behind the restarts and forgetting techniques in answer set programming. It is easy to prove a result similar to Theorem 16 for the graph SML_Π with *Restart* and *Forget LP* (for such graph a state is semi-terminal if no rule other than *Learn LP*, *Restart*, *Forget LP* is applicable to it.)

12.2 Extended Graph $\text{SML}_\Pi^\uparrow$

In this section we introduce an extended graph $\text{SML}_\Pi^\uparrow$ for the abstract answer set framework SML_Π similar as we introduced $\text{GTL}_{F,G}^\uparrow$ for $\text{GTL}_{F,G}$ in Section 10.3.

Recall the transition rule *Backjump LP* of SML_Π

$$\begin{array}{l}\textit{Backjump LP}:\\ P\ l^\Delta\ Q||\Gamma \implies P\ l'||\Gamma \ \text{ if } \left\{\begin{array}{l} P\ l^\Delta\ Q \text{ is inconsistent and}\\ \Pi \text{ entails } l'\vee\overline{P}.\end{array}\right.\end{array}$$

A state in the graph SML_Π is a *backjump state* if it is inconsistent, contains a decision literal, and is reachable from $\emptyset||\emptyset$. It turns out that *Backjump LP* is always applicable to a backjump state:

Theorem 17. *For a program* Π*, the transition rule Backjump LP is applicable to any backjump state in* SML_Π*.*

Theorem 17 guarantees that a backjump state in SML_Π is never semi-terminal. In the end of this section we show how Theorem 17 can be derived from the results proved later in this paper.

For a program Π, we say that a clause $l\vee C$ is a *reason* for l to be in a list of literals $P\ l\ Q$ with respect to Π, if Π entails $l\vee C$ and $\overline{C}\subseteq P$. We can equivalently restate the second condition of *Backjump LP* "Π entails $l'\vee\overline{P}$" as "there exists a reason for l' to be in $P\ l'$ with respect to Π" (note that $l'\vee\overline{P}$ is a reason for l' to be in $P\ l'$). We call a reason for l' to be in $P\ l'$ a *backjump clause*. Note that Theorem 17 asserts that a backjump clause always exists for a backjump state.

An *(extended) record* M relative to a program Π is a list of literals over the set of atoms occurring in Π where

(i) each literal l in M is annotated either by Δ or by a reason for l to be in M with respect to Π,

(ii) M contains no repetitions,

(iii) for any inconsistent prefix of M, its last literal is annotated by a reason.

For instance, let Π be the program

$$a \leftarrow \ not\ b$$
$$c.$$

The list of literals

$$b^{\Delta}\ a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}$$

is an extended record relative to Π. On the other hand, the lists of literals

$$a^{\Delta}\ \neg a^{\Delta} \qquad a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}\ b^{\Delta} \qquad b^{\Delta}\ a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}\ c^{\Delta}$$

are not extended records.

An *(extended) state* relative to a program Π is either a distinguished state *FailState* or a pair of the form $M||\Gamma$ where M is an extended record relative to Π, and Γ is the same as in the definition of an augmented state (i.e., Γ is a (multi)set of constraints formed from atoms occurring in Π that are entailed by Π.) For any extended state S relative to a program Π, the result of removing annotations from all nondecision literals of S is a state of SML_{Π}: we will denote this state by $S^{\downarrow}$.

For instance, consider program $a \leftarrow \ not\ b$. All pairs

$$\mathit{FailState} \quad \emptyset||\emptyset \qquad a^{\Delta}\ \neg\ b^{\neg b \vee \neg a}||\emptyset \qquad \neg a^{\Delta}\ b^{b \vee a}||\emptyset$$

are among valid extended states relative to this program. The corresponding states $S^{\downarrow}$ are

$$\mathit{FailState} \qquad \emptyset||\emptyset \qquad a^{\Delta}\ \neg\ b||\emptyset \qquad \neg a^{\Delta}\ b||\emptyset.$$

We now define a graph $\text{SML}^{\uparrow}_{\Pi}$ for any program Π. Its nodes are the extended states relative to Π. The transition rules of SML_{Π} are extended to $\text{SML}^{\uparrow}_{\Pi}$ as

follows: $S_1 \implies S_2$ is an edge in $\text{SML}^{\uparrow}_{\Pi}$ justified by a transition rule T if and only if $S_1^{\downarrow} \implies S_2^{\downarrow}$ is an edge in SML_{Π} justified by T.

The following lemma formally states the relationship between nodes of the graphs SML_{Π} and $\text{SML}^{\uparrow}_{\Pi}$:

Lemma 13. *For any program* Π*, if* S' *is a state reachable from* $\emptyset||\emptyset$ *in the graph* SML_{Π} *then there is a state* S *in the graph* $\text{SML}^{\uparrow}_{\Pi}$ *such that* $S^{\downarrow} = S'$*.*

The definitions of Basic transition rules and semi-terminal states in $\text{SML}^{\uparrow}_{\Pi}$ are similar to their definitions for SML_{Π}.

Theorem 16$^{\uparrow}$. *For any program* Π*,*

(a) every path in $\text{SML}^{\uparrow}_{\Pi}$ *contains only finitely many edges labeled by Basic transition rules,*

(b) for any semi-terminal state $M||\Gamma$ *of* $\text{SML}^{\uparrow}_{\Pi}$*,* M^+ *is an answer set of* Π*,*

(c) $\text{SML}^{\uparrow}_{\Pi}$ *contains an edge leading to FailState if and only if* Π *has no answer sets.*

We say that a state in the graph $\text{SML}^{\uparrow}_{\Pi}$ is a *backjump state* if its record is inconsistent and contains a decision literal. As in case of the graph SML_{Π}, any backjump state in $\text{SML}^{\uparrow}_{\Pi}$ is not semi-terminal:

Theorem 17$^{\uparrow}$. *For a program* Π*, the transition rule Backjump LP is applicable to any backjump state in* $\text{SML}^{\uparrow}_{\Pi}$*.*

Theorem 16 (b), (c) and Theorem 17 easily follow from Lemma 13 and Theorem 16$^{\uparrow}$ (b), (c) and Theorem 17$^{\uparrow}$ respectively. The proof of Theorem 16 (a) is similar to the proof of Theorem 16$^{\uparrow}$ (a).

12.3 Proofs of Theorem 16$^{\uparrow}$, Lemma 13, Theorem 17$^{\uparrow}$

12.3.1 Proof of Theorem 16$^{\uparrow}$

Lemma 14. *For any program* Π*, an extended record* M *relative to* Π*, and every assignment* X *such that* X^+ *is an answer set for* Π*, if* X *satisfies all decision literals in* M *then* $X \models M$*.*

Proof. By induction on the length of M. The property trivially holds for $\emptyset$. We assume that the property holds for any state with n elements. Consider any state M with $n+1$ elements. Let X be an assignment such that X^+ is an answer set for Π and X satisfies all decision literals in M. We will now show that $X \models M$.

Case 1. M has the form $P\ l^{\Delta}$. By the inductive hypothesis, $X \models P$. Since X satisfies all decision literals in M, $X \models l$.

Case 2. M has the form $P\ l^{l \vee C}$. By the inductive hypothesis, $X \models P$. By the definition of a reason, (i) Π entails $l \vee C$ and (ii) $\overline{C} \subseteq P$. From (ii) it follows that $P \models \neg C$. Consequently, $X \models \neg C$. From (i) it follows that for any assignment X such that X^+ is an answer set, $X \models l \vee C$. Consequently, $X \models l$. □

The proof of Theorem $16^{\uparrow}$ assumes the correctness of Theorem $17^{\uparrow}$ that we demonstrate in Section 12.3.3.

Theorem $16^{\uparrow}$. *For any program* Π*,*

(a) every path in $\text{SML}^{\uparrow}_{\Pi}$ *contains only finitely many edges labeled by Basic transition rules,*

(b) for any semi-terminal state $M||\Gamma$ *of* $\text{SML}^{\uparrow}_{\Pi}$*,* M^+ *is an answer set of* Π*,*

(c) $\text{SML}^{\uparrow}_{\Pi}$ *contains an edge leading to FailState if and only if* Π *has no answer sets.*

Proof. Parts (a) and (c) are proved as in the proof of Theorem $13^{\uparrow}$, using Lemma 14. (b) Let $M||\Gamma$ be a semi-terminal state so that none of the Basic rules are applicable. From the fact that *Decide* is not applicable, we conclude that M assigns all literals.

Furthermore, M is consistent. Indeed, assume that M is inconsistent. Then, since *Fail* is not applicable, M contains a decision literal. Consequently, $M||\Gamma$ is a backjump state. By Theorem $17^{\uparrow}$, the transition rule *Backjump LP* is applicable in $M||\Gamma$. This contradicts our assumption that $M||\Gamma$ is semi-terminal.

Also, M is a model of Π: since *Unit Propagate LP* is not applicable in $M||\Gamma$, it follows that for every rule $a \leftarrow B \in \Pi$, if $B \subseteq M$ then $a \in M$.

Assume that M^+ is not an answer set. Then, by Lemma 3, there is a non-empty unfounded set U on M with respect to Π such that $U \subseteq M$. It follows that *Unfounded* is applicable (with an arbitrary $a \in U$) in $M||\Gamma$. This contradicts the assumption that $M||\Gamma$ is semi-terminal.

□

12.3.2 Proof of Lemma 13

The proof uses the notion of loop formula [Lin and Zhao, 2002] (Section 9.1). In [Lee, 2005], the authors generalized the notion of a loop formula to arbitrary sets of atoms. We restate parts of their results here.

Given a set A of atoms by $Bodies(\Pi, A)$ we denote the set that consists of the elements of $Bodies(\Pi, a)$ for all a in A. Let Π be a program. For any set Y of atoms, the *external support formula* [Lee, 2005] for Y is

$$\bigvee_{B \in Bodies(\Pi,Y),\ B^+ \cap Y = \emptyset} B. \tag{12.2}$$

We will denote the external support formula by $ES_{\Pi,Y}$. For any set Y of atoms, the *loop formula* for Y is the implication

$$\bigvee_{a \in Y} a \rightarrow ES_{\Pi,Y}.$$

We can rewrite this formula as the disjunction

$$\bigwedge_{a \in Y} \neg a \vee ES_{\Pi,Y}. \tag{12.3}$$

From the *Main Theorem* in [Lee, 2005] we conclude:

Lemma on Loop Formulas. *For any program* Π*,* Π *entails loop formulas (12.3) for all sets* Y *of atoms that occur in* Π*.*

For a state S in the graph $\text{SML}_\Pi^\uparrow$, we say that $S^\downarrow$ in SML_Π is the *image* of S.

Lemma 13. *For any program* Π*, if* S' *is a state reachable from* $\emptyset||\emptyset$ *in the graph* SML_Π *then there is a state* S *in the graph* $\text{SML}_\Pi^\uparrow$ *such that* $S^\downarrow = S'$*.*

Proof. Since the property trivially holds for the initial state $\emptyset||\emptyset$, we only need to prove that all transition rules of SML_Π preserve it.

Consider an edge $M||\Gamma \Longrightarrow M'||\Gamma'$ in the graph SML_Π such that there is a state $M_1||\Gamma$ in the graph $\text{SML}_\Pi^\uparrow$ satisfying the condition $(M_1||\Gamma)^\downarrow = M||\Gamma$. We need to show that there is a state in the graph $\text{SML}_\Pi^\uparrow$ such that $M'||\Gamma'$ is its image

in SML_Π. Consider several cases that correspond to a transition rule leading from $M||\Gamma$ to $M'||\Gamma'$:

Unit Propagate LP:

$$M||\Gamma \implies M\ a||\Gamma \text{ if } \begin{cases} a \leftarrow B \in \Pi \text{ and} \\ B \subseteq M. \end{cases}$$

$M'||\Gamma'$ is $M\ a||\Gamma$. It is sufficient to prove that $M_1\ a^{a \vee \overline{B}}||\Gamma$ is a state of $\text{SML}_\Pi^\uparrow$. It is enough to show that a clause $a \vee \overline{B}$ is a reason for a to be in $M\ a$. By applicability conditions of *Unit Propagate LP*, $B \subseteq M$. Since Π entails its rule $a \leftarrow B$, Π entails $a \vee \overline{B}$.

All Rules Cancelled:

$$M||\Gamma \implies M\ \neg a||\Gamma \text{ if } \overline{B} \cap M \neq \emptyset \text{ for all } B \in \mathit{Bodies}(\Pi, a).$$

$M'||\Gamma'$ is $M\ \neg a||\Gamma$. Consider any $B \in \mathit{Bodies}(\Pi, a)$. Since $\overline{B} \cap M \neq \emptyset$, B contains a literal from $\overline{M}$: call it $f(B)$. It is sufficient to show that

$$\neg a \vee \bigvee_{B \in \mathit{Bodies}(\Pi, a)} f(B) \tag{12.4}$$

is a reason for $\neg a$ to be in $M\ \neg a$.

First, by the choice of $f(B)$, $f(B) \in \overline{M}$; consequently,

$$\overline{\bigvee_{B \in \mathit{Bodies}(\Pi, a)} f(B)} \subseteq M.$$

Second, since $f(B) \in B$, the loop formula $\neg a \vee ES_{\Pi,\{a\}}$ entails (12.4). By *Lemma on Loop Formulas*, it follows that Π entails (12.4).

Backchain True:

$$M||\Gamma \implies M\ l||\Gamma \text{ if } \begin{cases} a \leftarrow B \in \Pi, \\ a \in M, \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus \{B\}, \\ l \in B. \end{cases}$$

$M'||\Gamma'$ is $M\ l||\Gamma$. Consider any $B' \in \mathit{Bodies}(\Pi, a) \setminus B$. Since $\overline{B'} \cap M \neq \emptyset$, B' contains

a literal from $\overline{M}$: call it $f(B')$. A clause

$$l \vee \neg a \vee \bigvee_{B' \in Bodies(\Pi,a) \setminus B} f(B'). \tag{12.5}$$

is a reason for l to be in $M\ l$. The proof of this statement is similar to the case of *All Rules Cancelled*.

Backchain False λ:

$$M||\Gamma \implies M\ \overline{l}||\Gamma \ \text{ if } \begin{cases} a \leftarrow l, B\ \in\ \Pi \cup \Gamma, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M. \end{cases}$$

$M'||\Gamma'$ is $M\ \overline{l}||\Gamma$. A clause $\overline{l} \vee \overline{B} \vee a$ is a reason for $\overline{l}$ to be in $M\ \overline{l}$. The proof of this statement is similar to the case of *Unit Propagate LP*.

Unfounded:

$$M||\Gamma \implies M\ \neg a||\Gamma \ \text{ if } \begin{cases} M \text{ is consistent and} \\ a \in U \text{ for a set } U \text{ unfounded on } M \text{ with respect to } \Pi. \end{cases}$$

$M'||\Gamma'$ is $M\ \neg a||\Gamma$. Consider any $B \in Bodies(\Pi, U)$ such that $U \cap B^+ = \emptyset$. By the definition of an unfounded set, it follows that $\overline{B} \cap M \neq \emptyset$. Consequently, B contains a literal from $\overline{M}$: call it $f(B)$. The clause

$$\neg a \vee \bigvee_{Bodies(\Pi,U),\ B^+ \cap U = \emptyset} f(B) \tag{12.6}$$

is a reason for $\neg a$ to be in $M\ \neg a$. The proof of this statement is similar to the case of *All Rules Cancelled*.

Backjump LP, *Decide*, *Fail*, and *Learn LP*: obvious. □

The process of turning a state of SML_Π reachable from $\emptyset||\emptyset$ into a corresponding state of $\text{SML}_\Pi^\uparrow$ can be illustrated by the following example: Consider a

program Π

$$
\begin{array}{l}
a \leftarrow \mathit{not}\ b \\
b \leftarrow \mathit{not}\ a,\ \mathit{not}\ c \\
c \leftarrow \mathit{not}\ f \\
\leftarrow k,\ d \\
k \leftarrow l,\ \mathit{not}\ b \\
\leftarrow m,\ \mathit{not}\ l,\ \mathit{not}\ b \\
m \leftarrow\ \mathit{not}\ k,\ \mathit{not}\ l
\end{array}
\tag{12.7}
$$

and a path in SML_Π

$$
\begin{array}{l}
\emptyset||\emptyset \implies (\mathit{Decide}) \\
a^\Delta||\emptyset \implies (\mathit{All\ Rules\ Cancelled}) \\
a^\Delta\ \neg b||\emptyset \implies (\mathit{Decide}) \\
a^\Delta\ \neg b\ c^\Delta||\emptyset \implies (\mathit{Backchain\ True}) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f||\emptyset \implies (\mathit{Decide}) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f\ d^\Delta||\emptyset \implies (\mathit{Backchain\ False}\ \lambda) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f\ d^\Delta\ \neg k||\emptyset \implies (\mathit{Backchain\ False}\ \lambda) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f\ d^\Delta\ \neg k\ \neg l||\emptyset \implies (\mathit{Backchain\ False}\ \lambda) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f\ d^\Delta\ \neg k\ \neg l\ \neg m||\emptyset \implies (\mathit{Unit\ Propagate\ LP}) \\
a^\Delta\ \neg b\ c^\Delta\ \neg f\ d^\Delta\ \neg k\ \neg l\ \neg m\ m||\emptyset
\end{array}
\tag{12.8}
$$

The construction in the proof of Lemma 13 applied to the nodes in this path gives the following states of $\text{SML}^{\uparrow}_{\Pi}$:

$$
\begin{array}{l}
\emptyset||\emptyset \\
a^\Delta||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}\ d^\Delta||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}\ d^\Delta\ \neg k^{\neg k \vee \neg d}||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}\ d^\Delta\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}\ d^\Delta\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}||\emptyset \\
a^\Delta\ \neg b^{\neg b \vee \neg a}\ c^\Delta\ \neg f^{\neg f \vee \neg c}\ d^\Delta\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}\ m^{m \vee k \vee l}||\emptyset
\end{array}
\tag{12.9}
$$

It is clear that these nodes form a path in $\text{SML}^{\uparrow}_{\Pi}$ with every edge justified by the same transition rule as the corresponding edge in path (12.8) in SML_{Π}.

12.3.3 Proof of Theorem 17$^{\uparrow}$

In this section Π is an arbitrary and fixed logic program.

We say that a clause C is *conflicting* on a list M of literals if Π entails C, and $\overline{C} \subseteq lcp(M)$. For example, let M be the first component of the last state in (12.9):

$$a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ c^{\Delta}\ \neg f^{\neg f \vee \neg c}\ d^{\Delta}\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}\ m^{m \vee k \vee l}. \qquad (12.10)$$

Then, $lcp(M)$ is obtained by dropping the last element $m^{m \vee k \vee l}$ of M. It is clear that the reason $m \vee k \vee l$ for m to be in M is a conflicting clause on M.

Lemmas 10, 11, 12 hold for the case of extended record relative to a program. The proofs of the lemmas have to be modified only by replacing $F \wedge G$ with Π. Theorem 17$^{\uparrow}$ is proved as Theorem 14$^{\uparrow}$.

12.4 Decision and FirstUIP Backjumping and Learning for Answer Set Solvers

As we mentioned in Section 10.5, there are two common methods for describing a backjump clause construction in the SAT literature. The first one employes the implication graph [Marques-Silva and Sakallah, 1996a]. Ward and Schlipf [2004] extended the definition of an implication graph to the SMODELS algorithm and implemented the *FirstUIP* learning schema in the answer set solver SMODELS_{cc}. The second method used in SAT for characterizing a backjump clause derivation employes resolution. In Sections 10.4.3 and 10.5, we used $\text{GTL}^{\uparrow}_{F,G}$ formalism to describe the *BackjumpClause* and *BackjumpClauseFirstUIP* algorithms for computing *Decision* and *FirstUIP* backjump clauses respectively for the generate and test procedure. These algorithms follow the second tradition. It turns out that the *BackjumpClause* and *BackjumpClauseFirstUIP* algorithms can be also used for computing the ASP counterparts of *Decision* and *FirstUIP* backjump clauses. In fact, the *BackjumpClauseFirstUIP* algorithm is employed by the system SUP in its implementation of a conflict-driven backjumping and learning.

$lcp(M)$	$a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ c^{\Delta}\ \neg f^{\neg f \vee \neg c}\ d^{\Delta}\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}$
C_1	$m \vee k \vee l$
N	$\neg b^{\neg b \vee \neg a}\ \ \neg f^{\neg f \vee \neg c}\ \ \neg k^{\neg k \vee \neg d}\ \ \neg l^{\neg l \vee b \vee k}\ \ \neg m^{\neg m \vee l \vee b}$
R	$\neg b \vee \neg a, \quad \neg f \vee \neg c, \quad \neg k \vee \neg d, \quad \neg l \vee b \vee k, \quad \neg m \vee l \vee b$
C_2	$k \vee l \vee b$ is the resolvent of C_1 and $\neg m \vee l \vee b$
C_3	$k \vee b$ is the resolvent of C_2 and $\neg l \vee b \vee k$
C_4	$\neg d \vee b$ is the resolvent of C_3 and $\neg k \vee \neg d$
C_5	$\neg d \vee \neg a$ is the resolvent of C_4 and $\neg b \vee \neg a$

Figure 12.2: Sample execution of the *BackjumpClause* algorithm on backjump record (12.10) with respect to program (12.7).

For a program Π, we say that a record M is a *backjump record* with respect to Π if $M||\Gamma$ is a backjump state in $\textsc{sml}^{\uparrow}_{\Pi}$. The algorithms *BackjumpClause* and *BackjumpClauseFirstUIP* are applicable to the backjump records with respect to a program Π.

We first demonstrate an example of *BackjumpClause* application. Consider an execution of *BackjumpClause* on backjump record (12.10) with respect to program (12.7). Figure 12.2 illustrates what the values of $lcp(M)$, C, N, and R are during the execution of the *BackjumpClause* algorithm. By C_i we denote a value of C before the i-th iteration of the **while** loop. The algorithm terminates with the clause $\neg d \vee \neg a$. The proof of Theorem 17$^{\uparrow}$ asserts that (i) this clause is a backjump clause such that d and a are decision literals in M and (ii) the transition

$$\begin{array}{l} a^{\Delta} \neg b^{\neg b \vee \neg a}\ c^{\Delta} \neg f^{\neg f \vee \neg c}\ d^{\Delta} \neg k^{\neg k \vee \neg d} \neg l^{\neg l \vee b \vee k} \neg m^{\neg m \vee l \vee b}\ m^{m \vee k \vee l} || \emptyset \Longrightarrow \\ a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ \neg d^{\neg d \vee \neg a} || \emptyset \end{array} \tag{12.11}$$

in $\textsc{sml}^{\uparrow}{}_{\Pi}$ is an application of the transition rule *Backjump LP*. Indeed, by Lemma 12 $lcp(M)^{dec_M(\overline{\neg a})]}\ \neg\ d^{\neg d \vee \neg a}$, in other words $a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ \neg d^{\neg d \vee \neg a}$, is a record.

We now demonstrate an example of the *BackjumpClauseFirstUIP* application.

Consider an execution of the *BackjumpClauseFirstUIP* algorithm on backjump record (12.10) with respect to program (12.7). Figure 12.3 illustrates what the values of $lcp(M)$, C, N, and R are during the execution of the *BackjumpClause* algorithm. By C_i we denote a value of C before the i-th iteration of the **while** loop.

$lcp(M)$	$a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ c^{\Delta}\ \neg f^{\neg f \vee \neg c}\ d^{\Delta} \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}$
C_1	$m \vee k \vee l$
P	$d^{\Delta}\ \neg k^{\neg k \vee \neg d}\ \neg l^{\neg l \vee b \vee k}\ \neg m^{\neg m \vee l \vee b}$
R	$\neg k \vee \neg d, \quad \neg l \vee b \vee k, \quad \neg m \vee l \vee b$
C_2	$k \vee l \vee b$ is the resolvent of C_1 and $\neg m \vee l \vee b$
C_3	$k \vee b$ is the resolvent of C_2 and $\neg l \vee b \vee k$.

Figure 12.3: Sample execution of the *BackjumpClauseFirstUIP* algorithm on backjump record (12.10) with respect to program (12.7).

The *BackjumpClauseFirstUIP* algorithm terminates with the clause $k \vee b$. The proof of the correctness of *BackjumpClauseFirstUIP* asserts that (i) $k \vee b$ is a backjump clause and (ii) the transition

$$\begin{array}{l} a^{\Delta} \neg b^{\neg b \vee \neg a}\ c^{\Delta} \neg f^{\neg f \vee \neg c}\ d^{\Delta} \neg k^{\neg k \vee \neg d} \neg l^{\neg l \vee b \vee k} \neg m^{\neg m \vee l \vee b}\ m^{m \vee k \vee l} \Longrightarrow \\ a^{\Delta}\ \neg b^{\neg b \vee \neg a}\ k^{k \vee b} || \emptyset \end{array} \tag{12.12}$$

in $\text{SML}^{\uparrow}{}_{\Pi}$ is an application of *Backjump LP*.

12.5 Sup Algorithms

The work on $\text{SML}^{\uparrow}_{\Pi}$ that extends SM_{Π} with backjumping and learning facilitated the development of the answer set solver SUP [Lierler, 2008]. In Section 5.3 we demonstrated a method for specifying the algorithm of an answer set solver by means of the graph SM_{Π}. In particular, we described the SMODELS algorithm by assigning priorities to transition rules of SM_{Π}. In this section we use this method to describe the SUP algorithm by means of $\text{SML}^{\uparrow}_{\Pi}$. In the end of the section we will also use this framework to describe the algorithm of the answer set solver SMODELS_{cc} [Ward and Schlipf, 2004].

The system SUP assigns priorities to inference rules of $\text{SML}^{\uparrow}_{\Pi}$ as follows:

Backjump LP, Fail >>
Unit Propagate LP, All Rules Cancelled, Backchain True, Backchain False λ >>
Decide >>
Unfounded.

Also, SUP always applies the transition rule *Learn LP* in a non-semi-terminal state reached by an application of *Backjump LP*, because it implements conflict-driven backjumping and learning.[1]

For example, let Π be program (4.8). Path (12.1) corresponds to an execution of the system SUP.

In Section 12.4 we discuss details on which clause is being learned during the application of *Learn LP*. The system SUP implements *BackjumpClauseFirstUIP* procedure (Algorithm 4 in Section 10.5) to compute a backjump clause.

The strategy of SUP of assigning the transition rule *Unfounded* the lowest priority may be reasonable for many problems. For instance, it is easy to see that the transition rule *Unfounded* is redundant for tight programs. The SUP algorithm is similar to SAT-based answer set solvers such as ASSAT [Lin and Zhao, 2004] (see Section 9.2) and CMODELS [Giunchiglia *et al.*, 2006] (see Section 11.2) in the fact that it will first compute a supported model of a program and only then will test whether this model is indeed an answer set, i.e., whether *Unfounded* is applicable in this state.

We also note that SUP accepts input with choice and weight rules. Like CMODELS it uses transformations described in Section 8.1 to eliminate choice and weight rules in favor of semi-traditional rules. The graph $\text{SML}^{\uparrow}_{\Pi}$ can be easily generalized to semi-traditional programs.

In [Ward and Schlipf, 2004], the authors introduced the system SMODELS_{cc} that enhances the SMODELS algorithm with conflict-driven backjumping and learning. Here we use the graph $\text{SML}^{\uparrow}_{\Pi}$ to describe the SMODELS_{cc} algorithm. SMODELS_{cc} assigns priorities to inference rules of $\text{SML}^{\uparrow}_{\Pi}$ as follows[2]:

Backjump LP, *Fail* >>
Unit Propagate LP, *All Rules Cancelled*, *Backchain True*, *Backchain False* λ >>
Unfounded >>
Decide.

Note that the priority assignment that describes SMODELS_{cc} is different from that

[1] The system SUP (SMODELS_{cc}) also implements restarts and forgetting that may be modeled by the transition rules *Restart* and *Forget LP*. An application of these transition rules in $\text{SML}^{\uparrow}_{\Pi}$ relies on particular heuristics implemented by the solver.

[2] Its strategy for choosing a path in the graph $\text{SML}^{\uparrow}_{\Pi}$ is similar to that of SMODELS.

of SUP in the following: former assigns higher priority to *Unfounded* than *Decide*. Similarly to SUP, SMODELS$_{cc}$ always applies the transition rule *Learn LP* in a non-semi-terminal state reached by an application of *Backjump LP*. Unlike SUP that computes a backjump clause using the *BackjumpClauseFirstUIP* procedure based on clause resolution, SMODELS$_{cc}$ uses the extended notion of an implication graph introduced in [Ward and Schlipf, 2004] for this purpose.

For example, let Π be program (4.8). A path in $\text{SML}^{\uparrow}_{\Pi}$ from $\emptyset||\emptyset$ to the same semi-terminal node

$$\begin{array}{lll} \emptyset||\emptyset & \Longrightarrow & (\textit{Decide}) \\ a^{\Delta}||\emptyset & \Longrightarrow & (\textit{Unit Propagate LP}) \\ a^{\Delta}\ c||\emptyset & \Longrightarrow & (\textit{All Rules Cancelled}) \\ a^{\Delta}\ c\ \neg b||\emptyset & \Longrightarrow & (\textit{Unfounded}) \\ a^{\Delta}\ c\ \neg b\ \neg d||\emptyset & & \end{array}$$

corresponds to an execution of SMODELS$_{cc}$, but it does not correspond to any execution of system SUP because for the latter *Decide* is a rule of higher priority than *Unfounded*. On the other hand, path (12.1) does not correspond to any execution of SMODELS$_{cc}$ because for the latter *Unfounded* is a rule of higher priority than *Decide*.

We also note that SMODELS$_{cc}$ implementation accepts input with choice and cardinality constraint rules. Unlike SUP that eliminates these rules in favor of semi-traditional rules, SMODELS$_{cc}$ extends its inference mechanism directly to these rules. These extensions are out of the scope of this dissertation.

12.6 Implementation and Experimental Analysis

The implementation of SUP extends the SAT solver MINISAT (v1.12b). It utilizes

- the interface of the SAT solver MINISAT (v1.12b) that supports non-clausal constraints described in [Een and Sörensson, 2003b] in order to introduce additional inference possibilities, but unit propagation. In particular, SUP implements *Backchain True* and *All Rules Cancelled* by means of non-clausal constraints and it uses the unit propagate of MINISAT to capture *Unit Propagate LP* and *Backchain False*.

- parts of CMODELS code that eliminate weight and choice rules; perform model

Instance	SUP	CMODELS+MINISAT	SMODELS
pigeon.p9h8	1.32	0.56	4.8
pigeon.p10h9	9.24	6.28	47.19
pigeon.p11h10	113.78	85.02	509.27
pigeon.p12h11	t-o	t-o	t-o
15-puzzle.1	29.52	9.76	t-o
15-puzzle.2	205.78	45.55	t-o
15-puzzle.3	22.94	5.04	t-o
15-puzzle.4	367.26	93.93	t-o
15-puzzle.5	t-o	112.25	t-o
15-puzzle.6	t-o	127.45	t-o
15-puzzle.7	t-o	59.73	t-o
15-puzzle.8	24.75	3.46	374.39
15-puzzle.9	199.83	7.85	t-o

Figure 12.4: Tight Programs: Pigeon Hole, 15-Puzzle; traditional encoding; runtimes of SUP, CMODELS using MINISAT, and SMODELS.

verification; and compute loop formulas. In particular, SUP uses the latter two parts of CMODELS code to capture *Unfounded*.

As in Section 6.5.2 we describe here experiments conducted using the system whose technical specifications are presented in Section 6.5. In this section, we compare the performance of SUP version 0.4 with CMODELS using MINISAT, SMODELS, and SMODELS$_{cc}$. Details on the versions of the answer set solvers CMODELS, SMODELS, and SMODELS$_{cc}$ are provided in Section 6.5.

Figures 12.4 and 12.5 present the running times for SUP versus CMODELS using MINISAT and SMODELS on benchmarks described in Section 6.5.1: Pigeon Hole, 15-Puzzle, Graph Coloring, Schur Numbers, Putting Numbers, n-queens, Blocked n-queens. Programs encoding these problems are tight.

Figure 12.6 presents the running times for SUP versus CMODELS using MINISAT, SMODELS, and SMODELS$_{cc}$ on nontight programs that encode benchmarks described in Section 11.6: Deterministic Automaton, Wire Routing, Bounded Model Checking, Hamiltonian Cycle. Running times for the systems CMODELS using MINISAT, SMODELS, and SMODELS$_{cc}$ were reported previously. Here we report new results for running times of the answer set solver SUP. Overall the results demonstrated by SUP place the system in the class of efficient answer set solvers. It often outperforms

other native answer set solvers SMODELS and SMODELS$_{cc}$.

Figures 12.7 and 12.8 present the experimental results kindly provided to us by Martin Brain (March 7, 2010). The experiments were run using a 2.8Ghz Intel Xeon E5462 processor, running Scientific Linux 5.4. All solvers were built in 32 bit mode. Each test was limited to 3600 seconds of CPU time and 3Gb of RAM. The figures report running times of the answer set solvers CLASP (Section 14.1), CMODELS using MINISAT, and SUP used in the music composition tool ANTON.[3] ANTON is an automatic tool that composes melodic and harmonic Renaissance music in the style of the *Palestrina Rules*. Its core computational engine is an answer set solver. Figures 12.7 and 12.8 present running times for two different settings of ANTON: Rhythmic and Simple. The System SUP demonstrates competitive results as a computational engine of ANTON.

[3]http://www.cs.bath.ac.uk/ mjb/anton/

Instance	SUP	CMODELS+MINISAT	SMODELS
color.p1000.4	0.47	0.62	11.43
color.p6000.4	t-o	t-o	465.12
color.p3000.4	27.85	14.1	109.88
color.p3000.3	1.03	1.4	0.92
color.p6000.3	2.19	2.93	1.85
schur.p4n45	38.94	4.53	536.86
schur.p5n100	0.7	1.1	t-o
schur.p5n110	127.02	521.08	t-o
schur.p5n120	71.42	t-o	t-o
pn.gsquare-4-11-3-8	0.51	0.15	57.62
pn.gsquare-4-12-3-8	3.01	0.64	19.05
pn.gsquare-4-19-3-8	6.28	0.14	223.91
pn.gsquare-4-22-3-8	22.53	0.77	32.08
pn.gsquare-4-24-3-8	0.06	0.42	470.57
pn.gsquare-5-12-4-8	1.04	24.42	t-o
queens.q22	0.11	0.29	171.46
queens.q24	0.14	0.4	225.61
queens.q28	0.23	0.81	t-o
queens.q32	0.39	1.5	t-o
queens.q36	0.52	2.5	t-o
bqueens.50.1642398261	239.67	19.14	321.73
bqueens.50.1642399343	21.75	7.76	66.53
bqueens.50.1642399526	16.12	24.56	11.85
bqueens.50.1642400086	80.47	20.48	377.66
bqueens.50.1642401471	16.7	5.23	34.03
bqueens.50.1642402365	168.99	49.96	250.9
bqueens.50.1642402587	9.5	3.67	24.43
bqueens.50.1642403758	88.85	21.9	457.62
bqueens.50.1642404800	10.89	4.47	71.45
bqueens.50.1642405183	106.18	35.6	138.5

Figure 12.5: Tight Programs: Graph Coloring, Schur Numbers, Putting Numbers, n-queens, Blocked n-queens; encoding with choice rules and cardinality constraints; runtimes of SUP, CMODELS using MINISAT, and SMODELS.

Instance	SUP	CMODELS+MINISAT	SMODELS	SMODELS$_{cc}$
detA.Morin.mutex4	0.47	0.76	21.46	64.81
detA.Morin.phi5	0.78	0.23	1.17	4.43
detA.IDFD.mutex4	0.72	0.48	17.96	52.71
detA.IDFD.phi5	0.22	0.18	1.0	3.88
wire.10.x.10.b.5.a.25S	4.4	0.5	t-o	134.11
wire.10.x.10.b.5.a.35S	1.53	0.26	13.37	7.3
wire.10.x.10.b.5.a.20U	12.93	2.33	41.17	4.84
wire.12.x.12.b.5.a.15U	t-o	t-o	t-o	t-o
wire.12.x.12.b.5.a.20U	201.53	349.83	t-o	t-o
dp-8.fsa-D-i-O2-b8	0.02	0.05	1.51	0.72
dp-10.fsa-D-i-O2-b10	0.03	0.11	119.14	6.43
dp-12.fsa-D-i-O2-b9	114.87	159.57	454.57	t-o
hc-1S	1.11	0.55	t-o	36.35
hc-2S	31.01	2.29	t-o	16.09
hc-3S	2.94	8.98	t-o	t-o
hc-4S	2.07	1.66	0.82	3.78

Figure 12.6: Nontight Programs: Deterministic Automaton, Wire Routing, Bounded Model Checking, Hamiltonian Cycle; runtimes of SUP, CMODELS using MINISAT, SMODELS, and SMODELS$_{cc}$.

Style	Measures	Solvers		
		CLASP 1.3.2	CMODELS 3.79	SUP 0.4
9*solo	2-few	0.70	0.92	1.00
	2-many	1.37	1.65	1.40
	3-few	1.83	1.91	3.58
	3-many	6.37	3.23	6.23
	4-few	3.69	4.79	6.82
	4-many	37.62	8.81	9.13
	6-few	96.70	8.51	24.33
	6-many	1238.99	26.85	111.95
	8-few	295.33	29.47	98.74
	8-many	3374.40	51.95	379.28
9*duet	2-few	7.68	7.22	6.78
	2-many	14.09	20.70	20.17
	3-few	22.64	30.35	21.48
	3-many	245.06	150.66	205.02
	4-few	168.93	76.17	66.94
	4-many	2264.39	902.17	276.82
	6-few	3590.69	345.58	633.61
	6-many	m-o	2418.16	2198.27
	8-few	m-o	82.90	1669.23
	8-many	m-o	m-o	m-o
7*trio	2-few	25.16	13.39	39.05
	2-many	127.69	28.54	166.41
	3-few	283.15	48.90	264.53
	3-many	1336.46	3597.66	1235.31
	4-few	1794.29	397.64	766.25
	4-many	m-o	t-o	3305.92
	6-few	m-o	2467.05	2890.13
	6-many	m-o	m-o	m-o
7*quartet	2-few	325.96	45.87	91.69
	2-many	1192.52	141.55	859.42
	3-few	1019.19	204.64	296.06
	3-many	t-o	t-o	t-o
	4-few	t-o	2035.46	t-o
	4-many	m-o	m-o	m-o

Figure 12.7: ANTON, Rhythmic: runtimes of CLASP, CMODELS using MINISAT, and SUP.

Style	Measures	Solvers		
		CLASP 1.3.2	CMODELS 3.79	SUP 0.4
7.5*solo	2	0.09	0.19	0.10
	4	0.58	1.05	0.47
	6	1.42	2.23	1.21
	8	2.22	3.73	2.44
	10	4.69	5.68	3.86
	12	5.04	6.82	3.24
	14	10.88	8.77	5.79
	16	31.07	9.46	7.41
7.5*duet	2	0.24	0.56	0.28
	4	1.34	2.14	1.32
	6	6.80	4.69	5.50
	8	21.57	7.13	9.16
	10	70.89	29.33	27.04
	12	99.46	24.89	30.51
	14	202.59	41.59	133.34
	16	443.03	115.14	81.78
7.5*trio	2	0.44	0.96	0.57
	4	1.96	3.04	2.88
	6	6.17	5.48	7.56
	8	3.98	9.65	18.48
	10	116.60	11.67	12.60
	12	542.18	31.53	66.01
	14	862.54	37.66	62.94
	16	3364.77	53.54	132.23
7.5*quartet	2	0.86	1.65	1.43
	4	3.57	4.91	9.62
	6	128.33	11.89	8.08
	8	1083.83	24.07	75.78
	10	2778.41	81.35	127.57
	12	1784.37	201.12	75.09
	14	2801.68	347.58	259.19
	16	t-o	269.10	296.91

Figure 12.8: ANTON, Simple: runtimes of CLASP, CMODELS using MINISAT, and SUP.

Chapter 13

Extending Cmodels Algorithm to Disjunctive Programs

The answer set semantics for disjunctive logic programs was introduced in [Gelfond and Lifschitz, 1991]. Recall that in traditional programs rules have the form

$$a \leftarrow b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m$$

where $b_1, \ldots, b_m$ are atoms and a is an atom or symbol $\bot$. Disjunctive programs are composed of the rules that may contain disjunction of atoms

$$a_1 \vee \cdots \vee a_k$$

in the head. This is a special class of programs with nested expressions.

Disjunctive logic programs under the answer set semantics are more expressive than traditional programs. The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete [Eiter and Gottlob, 1993], while the same problem for a traditional program is NP-complete.

Until recently there were only two answer set systems that allowed programs with disjunctive rules DLV and GNT. The system DLV implements a specialized search algorithm tailored to find solutions for disjunctive answer set programs. The system GNT, on the other hand, implements the generate and test approach by using the answer set solver SMODELS to first *generate* candidate set of atoms, and then *test* this set whether it is indeed an answer set of a program. In this chapter we

introduce a SAT-based method for computing answer sets of a program. Similarly to the answer set solver GNT, this method will adopt generate and test approach. Unlike GNT, it will use a SAT solver for search in place of an answer set solver.

Section 13.1 starts this chapter by reviewing a class of disjunctive programs that is intermediate between semi-traditional programs (Section 7.5) and programs with nested expressions (Section 7.1). It also presents generalizations of completion, loop formulas, and the theorem on loop formulas to disjunctive programs. This paves the way to extending the SAT-based method for finding answer sets (Section 11.2) to disjunctive programs. Section 13.2 introduces a procedure for clausifying completion of a disjunctive program. Section 13.3 defines the CMODELS algorithm for disjunctive programs. In Section 13.4 we present a verification method for testing whether a model of completion is an answer set of a disjunctive program. In Section 11.3 we define the notion of a terminating loop for semi-traditional programs. Section 13.5 extends this notion to disjunctive programs and describes how such a loop can be computed. In Section 13.6 we present a proof to a theorem stated earlier in the chapter. Section 13.7 presents experimental analysis comparing the performance of CMODELS versus DLV and GNT.

13.1 Background: Disjunctive Programs

In this section we review the definitions of a disjunctive program and discuss the generalizations of completion, tightness, and loop formulas to such programs.

A disjunctive rule has the form

$$A \leftarrow B, \tag{13.1}$$

where its head A is a disjunction of atoms $a_1 \vee \cdots \vee a_k$ or $\bot$, and its body B is an expression

$$b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n, \tag{13.2}$$

where each b_i is an atom. Note that this rule is a special case of a rule with nested expressions introduced in Section 7.1. Also, Section 7.1 defines the reduct and answer set definitions for general programs with nested expressions and hence for programs with disjunctive rules.

We will write disjunctive rule (13.1) in one other form

$$A \leftarrow D, F, \tag{13.3}$$

where A is a disjunction of atoms $a_1 \vee \cdots \vee a_k$ or $\bot$; D is the positive part of the body

$$b_1, \ldots, b_l$$

and F is the negative part

$$not\ b_{l+1}, \ldots, not\ b_m, not\ not\ b_{m+1}, \ldots, not\ not\ b_n.$$

We identify the body of a rule (13.1) with the conjunction

$$b_1 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \ldots \neg b_m \wedge \neg\neg b_{m+1} \wedge \cdots \wedge \neg\neg b_n,$$

and we identify the rule itself with the clause

$$a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l \vee b_{l+1} \vee \cdots b_m \vee \neg b_{m+1} \vee \cdots \vee \neg b_n. \tag{13.4}$$

Lee and Lifschitz [2003] extended the notions of completion and loop formulas to disjunctive programs. The completion of a disjunctive program Π [Lee and Lifschitz, 2003], $Comp(\Pi)$, is defined as the set of propositional formulas that consists of the implication

$$B \rightarrow A \tag{13.5}$$

for every rule (13.1) in Π, and the implication

$$a \rightarrow \bigvee_{A \leftarrow B \in \Pi,\ a \in A} (B \wedge \bigwedge_{a' \in A \setminus \{a\}} \neg a') \tag{13.6}$$

for each atom a occurring in Π.

For instance, let Π be a program

$$\begin{array}{l} a \vee b \\ c \\ d \vee e \leftarrow a,\ c \\ d \leftarrow e \\ e \leftarrow d,\ not\ a. \end{array} \tag{13.7}$$

The completion $Comp(\Pi)$ follows[1]

$$\begin{array}{l} a \vee b \\ c \\ a \wedge c \rightarrow d \vee e \\ e \rightarrow d \\ d \wedge \neg a \rightarrow e \\ a \rightarrow \neg b \\ b \rightarrow \neg a \\ d \rightarrow (a \wedge c \wedge \neg e) \vee e \\ e \rightarrow (a \wedge c \wedge \neg d) \vee (d \wedge \neg a) \end{array}$$

The *dependency graph* of a disjunctive program Π is the directed graph G such that

- the vertices of G are the atoms occurring in Π
- for every rule (13.3) in Π, G has an edge from each atom in A to each atom in B .

Similarly to semi-traditional programs, a disjunctive program is *tight* [Lee and Lifschitz, 2003] if its dependency graph is acyclic.

For example, program (13.7) is not tight because it contains a cycle consisting of vertices d and e. On the other hand, if we drop the last rule from (13.7), the resulting program is tight.

In [Lee and Lifschitz, 2003], the authors extended Fages theorem to the case of tight disjunctive programs:

[1]We see $a \vee b$ as a shorthand for $\top \rightarrow a \vee b$ and c as a shorthand for $\top \rightarrow c$ and $c \rightarrow \top$.

Loop	Loop Formula
$\{a\}$	$a \rightarrow \neg b$
$\{b\}$	$b \rightarrow \neg a$
$\{c\}$	$c \rightarrow \top$
$\{d\}$	$d \rightarrow (a \wedge c \wedge \neg e) \vee e$
$\{e\}$	$e \rightarrow (a \wedge c \wedge \neg d) \vee (d \wedge \neg a)$
$\{e, d\}$	$e \vee d \rightarrow a \wedge c$

Figure 13.1: Loops and loop formulas for program (13.7)

Theorem 18 (Theorem on Tight Disjunctive Programs)**.** *For any tight disjunctive program* Π *and any set* X *of atoms,* X *is an answer set for* Π *if and only if* X *satisfies the completion of* Π*.*

The definition of a loop for a disjunctive program is identical to the definition of a loop for a semi-traditional program given in Section 9.1. For instance, program (13.7) contains 6 loops: $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{d, e\}$.

A *loop formula* F_L has the form

$$\bigvee L \rightarrow \bigvee R(L) \tag{13.8}$$

where $R(L)$ is the set of formulas

$$D \wedge F \wedge \bigwedge_{a \in A \setminus L} \neg a \tag{13.9}$$

for all rules (13.3) in Π such that $A \cap L \neq \emptyset$ and $D \cap L = \emptyset$ [Lee and Lifschitz, 2003].

As in the case of semi-traditional programs, by $LF(\Pi)$ we denote the set (conjunction) of all loop formulas for Π.

For instance, let Π be program (13.7). Its loop formulas are shown in Figure 13.1. $LF(\Pi)$ is the set of formulas in the right column. We note that as in the case of semi-traditional programs, for a singleton loop, the corresponding set of the bodies of some of the rules form right hand side of the implication (13.6) in the program's completion.

Theorem 19 (Theorem 1 from [Lee and Lifschitz, 2003])**.** *For any disjunctive program* Π *and any set* X *of atoms,* X *is an answer set for* Π *if and only if* X *satisfies* $Comp(\Pi) \cup LF(\Pi)$*.*

For instance, let Π be program (13.7). Its completion $Comp(\Pi)$ has three models: $\{b\ c\}$, $\{a\ c\ d\}$, and $\{b\ c\ d\ e\}$. The first two models satisfy $LF(\Pi)$. Consequently, they are also answer sets of Π. The model $\{b\ c\ d\ e\}$, on the other hand, does not satisfy the loop formula $e \vee d \rightarrow a \wedge c$ and therefore is not an answer set of Π.

13.2 Completion Clausification for Disjunctive Programs

The completion $Comp(\Pi)$ of a program Π converted to CNF using straightforward equivalent transformations can be exponentially larger than $Comp(\Pi)$. In this section we define a CNF formula $ED^{\vee}$-$Comp(\Pi)$ which will be a conservative extension of $Comp(\Pi)$ and will avoid exponential growth.

For a program Π, we construct an ED-set that consists of formulas containing explicit definitions for each body of each rule in Π whose

- head is not $\bot$, and
- body contains more than one atom.

We call such bodies *explicitly defined* by the ED-set.

For instance, let Π be program (13.7). The ED-set for Π consists of two explicit definitions $\{aux_1 \leftrightarrow a \wedge c,\ aux_2 \leftrightarrow d \wedge \neg a\}$ and the bodies of the fourth and sixth rules are explicitly defined by the ED-set.

For the completion $Comp(\Pi)$ of a program Π, we construct $Comp'(\Pi)$ by replacing the disjunctive terms in $Comp(\Pi)$ corresponding to the explicitly defined bodies of Π by their explicit definitions.

For example, let Π be program (13.7). Figure 13.2 presents the completion $Comp(\Pi)$ and $Comp'(\Pi)$.

For a program Π, a CNF formula $ED^{\vee}$-$Comp(\Pi)$ is a conjunction of

- $Comp'(\Pi)$ converted to CNF using the ED-transformation and
- formulas in ED-set converted to CNF using straightforward equivalent transformations.

$Comp(\Pi)$	$Comp'(\Pi)$
$a \vee b$	$a \vee b$
c	c
$a \wedge c \rightarrow d \vee e$	$aux_1 \rightarrow d \vee e$
$e \rightarrow d$	$e \rightarrow d$
$d \wedge \neg a \rightarrow e$	$aux_2 \rightarrow e$
$a \rightarrow \neg b$	$a \rightarrow \neg b$
$b \rightarrow \neg a$	$b \rightarrow \neg a$
$d \rightarrow (a \wedge c \wedge \neg e) \vee e$	$d \rightarrow (aux_1 \wedge \neg e) \vee e$
$e \rightarrow (a \wedge c \wedge \neg d) \vee (d \wedge \neg a)$	$e \rightarrow (aux_1 \wedge \neg d) \vee aux_2$

Figure 13.2: The completion $Comp(\Pi)$ and $Comp'(\Pi)$ where program Π is (13.7).

Let Π be program (13.7). Note that the ED-transformation on $Comp'(\Pi)$ will introduce two explicit definitions:

$$aux_3 \leftrightarrow aux_1 \wedge \neg e$$
$$aux_4 \leftrightarrow aux_1 \wedge \neg d.$$

$ED^{\vee}$-$Comp(\Pi)$ follows

	clausified explicite definitions
$a \vee b$	$\neg a \vee \neg c \vee aux_1$
c	$\neg aux_1 \vee a$
$\neg aux_1 \vee d \vee e$	$\neg aux_1 \vee c$
$\neg e \vee d$	$\neg d \vee a \vee aux_2$
$\neg aux_2 \vee e$	$\neg aux_2 \vee d$
$\neg a \vee \neg b$	$\neg aux_2 \vee \neg a$
$\neg b \vee \neg a$	$\neg aux_1 \vee d \vee aux_4$
$\neg d \vee aux_3 \vee e$	$\neg aux_3 \vee aux_1$
$\neg e \vee aux_4 \vee aux_2$	$\neg aux_3 \vee \neg e$
	$\neg aux_1 \vee e \vee aux_3$
	$\neg aux_4 \vee aux_1$
	$\neg aux_4 \vee \neg d$

13.3 Cmodels Algorithm for Disjunctive Programs

In this section we will define the CMODELS algorithm for disjunctive programs. We will use the graph $\text{GTL}^{\uparrow}_{F,G}$ for this purpose. Similarly to the case of semi-traditional programs, given a disjunctive program Π the CMODELS algorithm will generate models of $ED^{\vee}$-$Comp(\Pi)$ and then test these models whether they correspond to answer sets of Π. Recall that in the case of semi-traditional programs there is an efficient procedure that allows determining whether a model of the program's completion is an answer set. For disjunctive programs, deciding whether a model of $ED^{\vee}$-$Comp(\Pi)$ corresponds to some answer set of Π is co-NP-complete. This is not surprising as in general disjunctive programs allow to express all problems in the complexity class Σ^P_2 whereas semi-traditional programs allow to express all problems in the lower complexity class NP. Section 13.4 describes one of the possible methods for determining whether a model of the program's completion is an answer set.

The application of the CMODELS algorithm to a program Π can be viewed as constructing a path from $\emptyset||\emptyset$ to a semi-terminal node in $\text{GTL}^{\uparrow}_{ED^{\vee}\text{-}Comp(\Pi),LF(\Pi)}$. The CMODELS algorithm, like in case of semi-traditional nontight programs, assigns priorities to the inference rules of $\text{GTL}^{\uparrow}_{F,G}$ as follows:

$$
\begin{array}{l}
\textit{Backjump } GT, \textit{Fail} >> \\
\textit{Unit Propagate } \lambda >> \\
\textit{Decide} >> \\
\textit{Test}.
\end{array}
$$

The correctness of the CMODELS algorithm immediately follows from Theorem $13^{\uparrow}$.

In the case of semi-traditional programs there is an efficient procedure that allows determining whether the transition rule *Test* is applicable to a state $M||\Gamma$ when M is a model of the program's completion. As we mentioned earlier, for disjunctive programs deciding whether a model of $ED^{\vee}$-$Comp(\Pi)$ corresponds to some answer set of Π or, in other words, satisfies the loop formulas $LF(\Pi)$ of Π is co-NP-complete. Section 13.4 describes one of the possible methods for determining whether the transition rule *Test* is applicable to $M||\Gamma$. Section 13.5 extends the results on terminating loops (Section 11.3) for disjunctive programs that allows

defining the atomreason and loopformulareason methods for constructing a short reason C for a literal l given a state $M||\Gamma$ such that the transition rule *Test* is applicable to $M||\Gamma$ and $M||\Gamma \implies M\ l^C||\Gamma$ is the transition due to *Test*.

13.4 Verifying Models of Completion

In this section we discuss a verification method implemented in CMODELS for testing whether a model of completion is an answer set of a disjunctive program. Given a state $M||\Gamma$ in $\text{GTL}^{\uparrow}_{ED^{\vee}\text{-}Comp(\Pi),LF(\Pi)}$ such that M is a model of $ED^{\vee}$-$Comp(\Pi)$, one of the possible methods for determining whether the transition rule *Test* is applicable to $M||\Gamma$ relies on the minimality requirement of the definition of an answer set, i.e., set X of atoms is an answer set for a program Π if and only if X is a minimal set of atoms satisfying the reduct Π^X.

For any set M of literals, by M^- we denote the set of negative literals from M respectively. For instance, $\{a\ \neg b\}^-$ is $\{\neg b\}$. Let Π be a disjunctive program Π, and X be a model of $ED^{\vee}$-$Comp(\Pi)$. Recall that by X_Π we denote all literals in X whose atoms occur in Π. Note that X_Π is a model of $Comp(\Pi)$. We will denote X_Π by M. From Lemma 3i [Erdem and Lifschitz, 2003] it trivially follows that M^+ satisfies the reduct Π^{M^+}. Based on the definition of an answer set, if the formula $\Pi^{M^+} \wedge M^- \wedge \overline{M^+}$ is satisfiable then M^+ is not an answer set of Π. Indeed, if $\Pi^{M^+} \wedge M^- \wedge \overline{M^+}$ is satisfiable then there is a model M' of $\Pi^{M^+} \wedge M^- \wedge \overline{M^+}$ such that M'^+ is a proper subset of M^+ satisfying reduct Π^{M^+}. It follows that M is not minimal set satisfying Π^{M^+}.

We may now define a *minimality test procedure* on a program Π and a model of its completion M. A SAT solver is invoked on a clausified formula $\Pi^{M^+} \wedge M^- \wedge \overline{M^+}$. If the SAT solver determines that this formula is unsatisfied then the verified model M is indeed an answer set of Π. Otherwise, the solver returns some model M' of $\Pi^{M^+} \wedge M^- \wedge \overline{M^+}$ so that M'^+ is a proper subset of M^+ satisfying reduct Π^{M^+}. This minimality test procedure is similar to a procedure introduced in [Janhunen *et al.*, 2000] for the GNT system that encodes minimality test as an ASP problem and uses the ASP solver SMODELS to verify the minimality of a "perspective" model found by GNT.

In described minimality test procedure, a SAT solver is used for the model verification step. The idea of using a SAT solver for the task of verifying whether

a model of a program is an answer set is introduced in [Koch *et al.*, 2003]. In that work the concept of *unfounded-free* models of disjunctive programs is used in place of minimality in defining an algorithm for the model verification. Furthermore, the model verification algorithm in Figure 6 in [Koch *et al.*, 2003] improves the approach by taking advantage of

- a modularity property of a program so that verification is performed on parts of the program, and
- the fact that for the class of so called *head-cycle-free* programs the verification can be performed in polynomial time.

We also exploit these ideas in our implementation of minimality test procedure in CMODELS.

13.5 Terminating Loops for Disjunctive Programs

In Section 11.3 we defined the notion of a terminating loop for semi-traditional programs. Here we extend the notion of a terminating loop to disjunctive programs and state how such a loop can be computed.

Recall that by $G_{\Pi,X}$ we denote a subgraph of the dependency graph (Sections 4.5, 7.6) of a program Π induced by a set X of atoms.

Let Π be a disjunctive program, and X, X' be sets of atoms such that $X \models Comp(\Pi)$, $X' \subset X$, and $X' \models \Pi^X$. We understand that a loop is terminating under a set of atoms in the same way as in the case of semi-traditional programs.

The following theorem states that given a program Π and a set X of atoms such that $X \models Comp(\Pi)$ and X is not an answer set of Π, to calculate the loop formula of Π unsatisfied by X it is sufficient to

- find any proper subset X' of X such that $X' \models \Pi^X$ (minimality test procedure Section 13.4 finds such subset),
- compute a loop formula of any terminating loop in $G_{\Pi,X\setminus X'}$.

Theorem 15$^\vee$. *For a disjunctive program Π, and sets X, X' of atoms such that X is a models of $Comp(\Pi)$, $X' \subset X$, and X' is a model of Π^X, there is a terminating*

loop of Π *under* $X \setminus X'$*. Furthermore,* X *does not satisfy the loop formula of any terminating loop of* Π *under* $X \setminus X'$*.*

Given a state $M||\Gamma$ in $\text{GTL}^{\uparrow}_{\mathit{ED\text{-}Comp}^{\vee}(\Pi),LF(\Pi)}$ such that M is a model of $\mathit{ED\text{-}Comp}^{\vee}(\Pi)$ and *Test* is applicable to $M||\Gamma$, the atomreason approach requires a loop L such that for its loop formula F_L, $M \not\models F_L$. Theorem $15^{\vee}$ and the minimality test procedure provide us with a method for computing such L and F_L. The atomreason method for disjunctive programs is identical to atomreason method described in Section 11.2 for semi-traditional programs.

Given a state $M||\Gamma$ in $\text{GTL}^{\uparrow}_{\mathit{ED\text{-}Comp}^{\vee}(\Pi),LF(\Pi)}$ such that M is a model of $\mathit{ED\text{-}Comp}^{\vee}(\Pi)$ and *Test* is applicable to $M||\Gamma$, the loopformulareason method starts by finding a loop formula F_L of the form

$$\bigwedge_{l \in L} \overline{l} \vee \bigvee R(L) \tag{13.10}$$

so that $M \not\models F_L$. Recall that $R(L)$ denotes a formula

$$D \wedge F \wedge \bigwedge \neg a \tag{13.11}$$

where $D \wedge F$ stands for a body of some rule in the program and $\bigwedge \neg a$ stands for the conjunction of negation of some atoms occurring in the head of this rule. From the $\mathit{ED}^{\vee}\text{-}\mathit{Comp}(\Pi)$ construction it follows that if $D \wedge F$ contains more than one conjunctive term then $D \wedge F$ is explicitly defined and hence there is an auxiliary atom in $\mathit{ED}^{\vee}\text{-}\mathit{Comp}(\Pi)$ that stands for $D \wedge F$. By $\mathit{ED}^{\vee}(F_L)$ we denote F_L so that each occurrence of explicitly defined bodies of Π in F_L are replaced by their corresponding auxiliary atoms. We now apply the atomreason method on $\mathit{ED}^{\vee}(F_L)$ to construct a clause C. It is easy to see that for any literal $c \in C$, C is a reason for c to be in M c with respect to $\mathit{ED\text{-}Comp}^{\vee}(\Pi) \wedge LF(\Pi)$. The system CMODELS with the default settings will only consider edges due to the transition rule *Test* of the kind $M||\Gamma \implies M\ c^{C}||\Gamma$. This concludes the description of the loopformulareason method.

13.6 Proof of Theorem 15$^\vee$

We say that a rule $A \leftarrow B \in \Pi$ is *supporting* atom a under set X of atoms if $A \cap X = \{a\}$, and $X \models B$.

Lemma 15. *For a disjunctive program* Π*, and a model* X *of* $Comp(\Pi)$*, if* $a \in X$ *then there must be a supporting rule* $A \leftarrow B$ *in* Π *for* a *under* X*.*

Proof. From the completion construction there must be a clause

$$a \rightarrow \bigvee_{A \leftarrow B \in \Pi,\ a \in A} (B \wedge \bigwedge_{a' \in A \setminus \{a\}} \neg a')$$

in $Comp(\Pi)$. This clause is satisfied by X and therefore there exists at least one rule $A \leftarrow B$ such that $a \in A$, $X \models B$, and $A \cap X = \{a\}$. Such rule is a supporting rule for a under X. □

Recall that disjunctive rule (13.1) can be written in the form

$$A \leftarrow D, F, \tag{13.12}$$

where A is the head, D is the positive part of the body, and F is the negative part of the body.

Lemma 16. *For a disjunctive program* Π*, sets* X*,* X' *of atoms such that* X *is a model of* $Comp(\Pi)$*,* $X' \subset X$*, and* X' *is a model of* Π^X*, any supporting rule* $A \leftarrow D, F$ *for an atom in* $X \setminus X'$ *under* X *is such that* $D \cap (X \setminus X') \neq \emptyset$*.*

Proof. Consider any atom $a \in X' \setminus X$. Suppose there exists a supporting rule $A \leftarrow D, F$ for a under X such that $D \cap (X \setminus X') = \emptyset$, By the definition of a supporting rule, $X \models D, F$. It follows that $D \subseteq X'$. Furthermore, rule $A \leftarrow D, F^X \in \Pi^X$, where F^X is a conjunction of $\top$'s. We are given that $X' \models \Pi^X$, hence $X' \models A \leftarrow D, F^X$. Since $D \subseteq X'$, $X' \models D, F^X$ hence $X' \models A$ and $a \in X'$. This contradicts the assumption that $a \in X \setminus X'$. □

Recall that for a set X of atoms, by $G_{\Pi,X}$ we denote a subgraph of the dependency graph of a program Π induced by X.

Lemma 17. *For a disjunctive program* Π*, sets* X*,* X' *of atoms such that* X *is a model of* $Comp(\Pi)$*,* $X' \subset X$*,* X' *is a model of* Π^X*, for any atom* $a \in X \setminus X'$*, there is a strongly connected component* $L \subseteq X \setminus X'$ *in* $G_{\Pi, X \setminus X'}$ *and for some* $b \in L$*, there is a directed path from* a *to* b *in* $G_{X \setminus X'}$*.*

Proof. From Lemma 15 each atom $a \in X \setminus X'$ has a supporting rule under X. From Lemma 16 it follows that each supporting rule $A \leftarrow D, F$ for a is such that $D \cap (X \setminus X') \neq \emptyset$. Consequently, there is an atom $b \in D$ so that $b \in X \setminus X'$. From the construction of $G_{\Pi, X \setminus X'}$ for each atom $a \in X \setminus X'$ there must be an arc (a, b) where $b \in X \setminus X'$. Since the atom a is any node and $G_{\Pi, X \setminus X'}$ has finite number of nodes, we derive that there is a cycle in $G_{\Pi, X \setminus X'}$ and thus there is a strongly connected component L reachable from a. □

Theorem 15$^\vee$. *For a disjunctive program* Π*, and sets* X*,* X' *of atoms such that* X *is a model of* $Comp(\Pi)$*,* $X' \subset X$*, and* X' *is a model of* Π^X*, there is a terminating loop of* Π *under* $X \setminus X'$*. Furthermore,* X *does not satisfy the loop formula of any terminating loop of* Π *under* $X \setminus X'$*.*

Proof. From Lemma 17 it follows that there exists a strongly connected component in $G_{\Pi, X \setminus X'}$. Clearly, if there exists a strongly connected component in $G_{\Pi, X \setminus X'}$ then there exists a terminating loop.

Assume L is a terminating loop of Π under $X \setminus X'$. Its loop formula is of the form (13.8). Suppose that the model X satisfies the loop formula of L. It follows that there there is a rule of the form (13.3) — $A \leftarrow D, F$ — where $A \cap L \neq \emptyset$, and $D \cap L = \emptyset$, such that

$$X \models D, F \wedge \bigwedge_{a \in A \setminus L} \neg a \tag{13.13}$$

Consider this rule.

Case 1. $(X \setminus X') \cap D \neq \emptyset$. Let b be an atom such that $b \in (X \setminus X') \cap D$. By the condition $D \cap L = \emptyset$, $b \notin L$. By Lemma 17 there must be a strongly connected component L' such that there is a directed path from b to some atom in L'. Since $b \notin L$, $L' \neq L$. Take any atom c in $A \cap L$. From $G_{\Pi, X \setminus X'}$ construction, c has an edge to b, and hence has a directed path to some atom in L'. This contradicts to our assumption that L is terminating.

Case 2. $(X \setminus X') \cap D = \emptyset$. From (13.13) we conclude that $D \subseteq X'$ and $A \leftarrow D, F^X \in \Pi^X$, where F^X is the conjunction of $\top$. We are given that $X' \models \Pi^X$. Consequently, $X' \models A \leftarrow D, F^X$. Since $D \subseteq X'$, $X' \models A$. From (13.13) and the fact that $X' \subset X$ it follows that $X' \models \bigwedge_{a \in A \setminus L} \neg a$. It follows that $X' \cap A \cap L \neq \emptyset$. This contradicts to our assumption that $L \subseteq X \setminus X'$. □

13.7 Experimental Analysis

As in Section 6.5.2 we describe here experiments that were conducted using the system whose technical specifications are presented in Section 6.5. In this section, we compare the performance of CMODELS using MINISAT with DLV and GNT version 2. Details on the versions of answer set solvers CMODELS and DLV are provided in Section 6.5. We note that CMODELS uses the SAT solver ZCHAFF for minimality testing in its implementation.

The disjunctive benchmarks that we used are *Strategic Companies* and *2QBF*. Here are their brief descriptions[2]:

- In *Strategic Companies* a holding owns companies that produces some goods. Several companies may have joint control over another company. Some of these companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding after the transaction. A company is *strategic*, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. Given two companies a and b, the problem is to compute a strategic set containing both a and b, or determine that no such set exists.

- In *2QBF* the problem is to decide whether a quantified boolean formula F defined as

  $$\forall \mathbf{y} \exists \mathbf{x} G$$

  (where $\mathbf{x}$ and $\mathbf{y}$ are disjoint sets of variables and G is a CNF formula over variables from $\mathbf{x} \cup \mathbf{y}$) is true.

Both problems are Σ_2^p-hard.

[2]These descriptions follow http://www.cs.engr.uky.edu/ai/benchmarks.html .

Instance	LPARSE	CMODELS		DLV	GNT
		MINISAT	ZCHAFF		
strategic.250.a	0.08	0.05	0.02	0.03	0.16
strategic.250.b	0.08	0.04	0.02	0.03	0.13
strategic.1000.a	0.34	0.26	0.17	0.14	2.08
strategic.1000.b	0.34	0.24	0.15	0.15	2.39
strategic.2500.a	0.89	0.73	0.7	0.72	18.03
strategic.2500.b	0.89	0.72	0.74	0.71	16.43
strategic.6500.a	2.26	2.21	6.26	4.58	111.57
strategic.6500.b	2.35	2.29	1.7	4.54	104.96
strategic.9500.a	3.37	4.0	28.96	9.8	228.23
strategic.9500.b	3.38	4.2	1.83	9.71	221.28
2qbf3dnf1	0.01	0.0	0.0	0.2	t-o
2qbf3dnf2	0.13	276.31	11.49	2.23	t-o
2qbf3dnf3	0.13	279.05	65.73	1.41	t-o
2qbf3dnf4	0.84	t-o	159.7	3.92	t-o
2qbf3dnf5	0.14	0.15	0.1	0.89	t-o
2qbf3dnf6	0.05	0.01	0.01	0.07	t-o
2qbf3dnf7	0.02	0.01	0.0	0.01	t-o
2qbf3dnf8	0.01	0.47	0.01	4.71	t-o
2qbf3dnf9	0.01	0.0	0.0	0.08	2.66
random2qbf0.8-79-1	0.02	152.5	363.7	2.86	t-o
random2qbf0.8-79-2	0.02	14.67	17.91	0.07	t-o
random2qbf0.8-79-3	0.02	277.47	416.63	2.84	t-o
random2qbf0.8-79-4	0.03	83.22	282.44	0.19	t-o
random2qbf0.8-79-5	0.02	t-o	t-o	5.08	t-o
random2qbf0.8-79-6	0.02	48.67	359.49	2.33	t-o
random2qbf0.8-79-7	0.02	582.1	t-o	0.02	t-o
random2qbf0.8-79-8	0.03	35.69	125.34	0.17	t-o
random2qbf0.8-79-9	0.02	12.6	40.66	0.1	t-o
random2qbf0.8-79-10	0.02	49.92	142.05	0.74	t-o

Figure 13.3: Strategic Company, 2QBF, Random 2QBF; runtimes of LPARSE, CMODELS using MINISAT, CMODELS using ZCHAFF, DLV,GNT.

Figure 13.3 reports the performance of the grounder LPARSE, CMODELS using MINISAT, CMODELS using ZCHAFF, DLV, and GNT. We may note that on all instances CMODELS performs better than GNT. Group of 2QBF instances whose name starts with "random" is randomly generated. On these instances DLV outperforms CMODELS

by the order of magnitude suggesting that on such instances the "native" approach is superior to the SAT-based approach. Nevertheless, on other instances CMODELS often demonstrates competitive results.

Chapter 14

Related Work

The two SAT-based answer set solvers, CMODELS discussed in this dissertation and ASSAT designed by Lin and Zhao [2002], were developed independently and simultaneously on the basis of an earlier publication [Babovich *et al.*, 2000]. After the first publications on CMODELS [Lierler and Maratea, 2004; Giunchiglia *et al.*, 2004b], three research groups created the other systems SAG, PBMODELS, and CLASP, which enhanced the approach of ASSAT and CMODELS. The new systems are in some ways more sophisticated and efficient. In this chapter we briefly describe the ideas behind these systems.

14.1 Sag and Clasp

The Systems SAG [Lin *et al.*, 2006] and CLASP [Gebser *et al.*, 2007b] are answer set solvers for nondisjunctive programs that are enhancements of CMODELS and they are similar to CMODELS in several ways. First, they compute and clausify the program's completion and then use unit propagate on resulting propositional formula as an inference mechanism. Second, they guide their search by means of loop formulas. Third, they implement conflict-driven backjumping and learning. Also, SAG uses SAT-solvers for search. The systems differ from CMODELS in the following ways

- they maintain a data structure representing the input logic program throughout the whole computation,
- in addition to implementing inference rules of the graph $\text{GTL}_{F,G}$ they also

implement the inference rule *Unfounded* of SM_Π. A hybrid graph combining the inference rule *Unfounded* of SM_Π and the inference rules of $\text{GTL}_{F,G}$ may be used to describe the SAG and CLASP algorithms.

The system SAG assigns the same priorities to the inference rules of the hybrid graph as CMODELS. Also, SAG at random decides whether to apply the inference rule *Unfounded* in a state.

On the other hand, the system CLASP assigns priorities to the inference rules of the hybrid graph as follows:

Backjump GT, *Fail* >>
Unit Propagate λ, *Unfounded* >>
Decide.

Like CMODELS, both SAG and CLASP always apply the transition rule *Learn GT* in a non-semi-terminal state reached by an application of *Backjump GT*.

In the experimental analysis in [Lin *et al.*, 2006], the authors demonstrated that SAG usually performs at least as well as CMODELS, and on some benchmarks it is by an order of magnitude faster.

The system CLASP was originally inspired by the work in SAT-based answer set solving that used the notion of completion, loop formulas as its base computation means. CLASP approach proved to be successful. In *The First Answer Set Programming System Competition*, 2007 (see Section 11.7) out of ten participating answer set solvers CLASP was a winner in two out of three qualifying tracks and took the third place in the third track. CMODELS, on the other hand, took the second place in the latter track. In *The Second Answer Set Programming System Competition*, 2009 (see Section 11.7) out of sixteen participating answer set solvers a portfolio solver based on CLASP was the winner in the Desicion Problems track. CMODELS took the second place in this track. In Figure 14.1, we present running times of the system CLASP version 1.3.2 versus CMODELS using MINISAT and SMODELS on non-tight programs encoding Deterministic Automaton, Wire Routing, Bounded Model Checking, Hamiltonian Cycle benchmarks. We conducted experiments using the system and solvers whose technical specifications are presented in Section 6.5. Details on the versions of CMODELS and SMODELS are provided in Section 6.5.

Instance	CLASP	CMODELS+MINISAT	SMODELS
detA.Morin.mutex4	0.26	0.76	21.46
detA.Morin.phi5	0.05	0.23	1.17
detA.IDFD.mutex4	0.23	0.48	17.96
detA.IDFD.phi5	0.04	0.18	1.0
wire.10.x.10.b.5.a.25S	1.55	0.5	t-o
wire.10.x.10.b.5.a.35S	0.27	0.26	13.37
wire.10.x.10.b.5.a.20U	0.1	2.33	41.17
wire.12.x.12.b.5.a.15U	20.77	t-o	t-o
wire.12.x.12.b.5.a.20U	29.69	349.83	t-o
dp-8.fsa-D-i-O2-b8	0.02	0.05	1.51
dp-10.fsa-D-i-O2-b10	0.32	0.11	119.14
dp-12.fsa-D-i-O2-b9	78.56	159.57	454.57
hc-1S	41.67	0.55	t-o
hc-2S	25.45	2.29	t-o
hc-3S	0.39	8.98	t-o
hc-4S	158.34	1.66	0.82

Figure 14.1: Nontight Programs: Deterministic Automaton, Wire Routing, Bounded Model Checking, Hamiltonian Cycle; runtimes of CLASP, CMODELS using MINISAT, and SMODELS.

Nevertheless, CLASP misses one of the main postulates of this work where we advocate the reusing and leveraging on already existing computational technology.

14.2 Pbmodels – Weight Rules via Pseudoboolean Solvers

Liu and Truszczyński [2005a] designed and implemented the system PBMODELS to compute answer sets of logic programs with weight rules using, in part, ideas discussed in Sections 9.2, 10.1, and 11.2. The approach discussed in Chapter 7 involves translating programs with weight rules into propositional logic. PBMODELS investigates another possibility by proposing the translation of logic programs with weight rules into the *extended* propositional logic that allows weight atoms [Liu and Truszczyński, 2005b]. Unlike CMODELS or ASSAT that use classical SAT-solvers, PBMODELS uses specialized solvers that accept extended propositional formulas such as SATZOO [Eén and Sörensson, 2003a], PBS [Aloul *et al.*, 2003], WSATOIP [Walser, 2007], and WSATCC [Liu and Truszczyński, 2003]. We call such solvers pseudo-

boolean.

Liu and Truszczyński [2005a] extended the results on completion and loop formulas from [Lin and Zhao, 2004] to programs with weight rules and to extended propositional logic. These findings allowed PBMODELS to implement procedure similar to the one of ASSAT using pseudoboolean solvers mentioned above for search.

The approach discussed in Chapter 7 that allows compiling away weight rules may lead to significantly larger programs and corresponding propositional theories. PBMODELS, on the other hand, avoids the growth of the resulting propositional theory by using a more expressive target language. Afterwards, PBMODELS requires more specialized search procedures than traditional DPLL.

In *The First Answer Set Programming System Competition* (see Section 11.7), PBMODELS proved to be a competitive participant by gaining the second place in two out of three qualifying tracks.

Chapter 15

Conclusions

In this dissertation we have proposed and studied a SAT-based method for computing answer sets of a program. We have developed a graph-based theoretical framework that is well-suited for describing, proving correctness, and comparing algorithms underlying native and SAT-based answer set solvers. We have implemented and evaluated the SAT-based answer set solver CMODELS and the native answer set solver SUP that rely on theoretical findings described in the dissertation.

For tight programs, a straightforward method for computing answer sets using SAT solvers is available: it suffices to compute the program's completion and enumerate its models using a SAT solver. The first version of CMODELS which implemented this method was made publicly available in 2003. It used a simplification procedure that sometimes allowed us to eliminate parts of a given program and as a result to shorten the CNF representation of its completion.

In the next version, CMODELS was extended to choice and weight rules, which can be eliminated in favor of semi-traditional rules. The translation of an extended program to a program with semi-traditional rules adopted by CMODELS may introduce a large number of new atoms and rules. Nevertheless, experimental analysis suggests that it is still a viable approach in comparison with the native search procedure implemented in SMODELS.

Extending the SAT-based method to computing answer sets of nontight programs in an efficient manner was one of the main contributions of this work. The straightforward approach of adding all loop formulas to the program's completion is not feasible, because the number of loop formulas of a program may be expo-

nentially large. The SAT-based answer set solver ASSAT addresses this difficulty by invoking a SAT solver multiple times, with new loop formulas added each time until an answer set is found. The main drawback of the ASSAT method is that all information gained during the earlier invocations of a SAT solver is lost and hence the same parts of the search tree may be investigated many times. Typically, modern SAT solvers implement conflict-driven learning that is one of the most efficient techniques developed in propositional satisfiability. In this dissertation we designed a method of using loop formulas that takes advantage of this advanced feature. Unlike ASSAT, which treats the SAT solver as a black box, our system CMODELS modifies the SAT solver procedure to allow it to learn clauses based on loop formulas on demand. On the one hand, by slightly modifying a SAT solver computation we ensure that it is invoked only once. On the other hand, employing conflict-driven learning to learn information based on loop formulas allows us to guide search of a SAT solver using valuable constraints provided by loop formulas. The implementation of this method in CMODELS demonstrated competitive results, proving the effectiveness of the SAT-based approach.

During the course of the dissertation, we proposed and implemented the SUP algorithm for finding answer sets that can be seen as a combination of computational ideas behind CMODELS and SMODELS. Like CMODELS, the solver SUP operates by computing a sequence of supported models of the given program, but it does not form the completion. Instead, SUP runs the *Atleast* algorithm, one of the main building blocks of the SMODELS procedure.

In this dissertation we showed how to model algorithms for computing answer sets of a program by means of simple mathematical objects, graphs. We built upon the abstract framework for describing DPLL-like algorithms to capture the computation performed by native and SAT-based ASP algorithms. We characterized in this way the algorithms of the SAT-based answer set solver CMODELS and the native answer set solvers SMODELS, SUP, and SMODELS$_{cc}$. This approach simplifies the analysis of the correctness of algorithms. For instance, we used it to demonstrate the correctness of the answer set solvers designed in the course of the work on this dissertation: CMODELS and SUP. Furthermore, the abstract framework method allows us to study the relationship between various algorithms using the structure of the corresponding graphs. For example, we used this method to establish that applying the SMODELS algorithm to a tight program essentially amounts to applying

DPLL to its completion. Also, the description of the SUP and SMODELS$_{cc}$ algorithms using the graph SML$_{\Pi}$ reflects the differences between them in a simple manner via distinct assignments of priorities to the edges of the graph that characterize these systems. The work on this abstract framework helped us design the solver SUP.

The abstract framework provided a convenient tool for specifying algorithms for computing backjump clauses used in conflict-driven backjumping and learning for both SAT-based and native answer set solvers.

The ideas presented in this dissertation have led other researchers to designing several successful systems for computing answer sets: SAG, PBMODELS, and CLASP. This fact provides additional evidence in favor of the SAT-based approach to answer set programming.

Bibliography

[Aloul *et al.*, 2003] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of SAT-2003*, page 346 353, 2003.

[Armando *et al.*, 2004] A. Armando, L. Compagna, and Yu. Lierler. Automatic compilation of protocol insecurity problems into logic programming. In *Proceedings of 9th European Conference in Logics in Artificial Intelligence (JELIA-04)*, Lecture Notes In Artificial Intelligence, pages 617–627. Springer, 2004.

[Aura *et al.*, 2000] Tuomas Aura, Matt Bishop, and Dean Sniegowski. Analizing single-server network inhibition. In *Proceedings of the 13th EEEI Computer Security Foundation Workshop*, pages 108–117, 2000.

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[1] In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2000.

[Balduccini and Gelfond, 2003] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with a-prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.

[Baptista and Marques-Silva, 2000] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *6th CP*, page 489494, 2000.

[Baral and Uyan, 2001] Chitta Baral and Cenk Uyan. Declarative specification and

[1] `http://arxiv.org/abs/cs.ai/0003042` .

solution of combinatorial auctions using logic programming. *Lecture Notes in Computer Science*, 2173:186–199, 2001.

[Baral *et al.*, 2005] Chitta Baral, Gregory Gelfond, Michael Gelfond, and Richard Scherl. Textual inference by combining multiple logic programming paradigms. In *AAAI Workshop on Inference for Textual Question Answering*, 2005.

[Bayardo and Schrag, 1997] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 203–208, 1997.

[Beame *et al.*, 2004] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Artificial Intelligence Research*, 22:319–351, 2004.

[Borchert *et al.*, 2004] P. Borchert, C. Anger, T. Schaub, and M. Truszczyński. Towards systematic benchmarking in answer set programming: The Dagstuhl initiative. In *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, pages 3–7, 2004.

[Brain *et al.*, 2006] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying answer set programming to superoptimisation. In Sandro Etalle and Mirosław Truszczynski, editors, *Logic Programming*, LNCS 4079, pages 270–284. Springer, 2006.

[Brooks *et al.*, 2007] Daniel R. Brooks, Esra Erdem, Selim T. Erdoğan, James W. Minett, and Donald Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39:471–511, 2007.

[Buccafurri *et al.*, 1997] Francesco Buccafurri, Nicola Leone, and Pasquale Pasquale Rullo. Adding weak constraints to disjunctive datalog. In *Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, 1997.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Cormen *et al.*, 1994] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1994.

[Ştefănescu *et al.*, 2003] A. Ştefănescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *In* Proceedings CONCUR*, LNCS 2761.*, 2003.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Dell'Armi *et al.*, 2003] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, 2003.

[Denecker *et al.*, 2009] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczynski. The second answer set programming system competition.[2] In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In Sam Steel and Rachid Alami, editors, *Proceedings of European Conference on Planning*, pages 169–181. Springer, 1997.

[Dixon *et al.*, 2004] H. Dixon, M. Ginsberg, E. Luks, and A. Parkes. Generalizing boolean satisfiability ii: Theory. *Journal of Artificial Inteligence Research (JAIR)*, 2004.

[Dowling and Gallier, 1984] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1984.

[Een and Biere, 2005] Niklas Een and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, 2005.

[Eén and Sörensson, 2003a] N. Eén and N. Sörensson. An extensible sat solver. In *Proceedings of SAT-2003*, pages 502–518, 2003.

[2] `http://www.cs.kuleuven.be/~dtai/events/asp-competition/paper.pdf` .

[Een and Sörensson, 2003b] Niklas Een and Niklas Sörensson. An extensible sat-solver. In *SAT*, 2003.

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In Dale Miller, editor, *Proceedings of International Logic Programming Symposium (ILPS)*, pages 266–278, 1993.

[Eiter *et al.*, 1997] Thomas Eiter, Nicola Leone, Christinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for non-monotonic reasoning. In *Proceedings of the 4th International Conference on Logic Programming and Non-monotonic Reasoning*, pages 363–374. Springer, 1997.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 406–417, 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Diagnosis frontend of the DLV system. *The European Journal of Artificial Intelligence*, 12(1–2):99–111, 1999.

[Erdem and Lifschitz, 2001] Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 242–254, 2001.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Erdem and Wong, 2004] E. Erdem and M.D.F. Wong. Rectilinear steiner tree construction using answer set programming. In *Proceedings of International Conference on Logic Programming (ICLP'04)*, pages 386–399, 2004.

[Fages, 1994] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[Gebser and Schaub, 2007] Martin Gebser and Torsten Schaub. Generic tableaux for answer set programming. In *Proceedings of 23d International Conference on Logic Programming (ICLP'07)*, pages 119–133. Springer, 2007.

[Gebser *et al.*, 2007a] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 266–271, 2007.

[Gebser *et al.*, 2007b] Martin Gebser, Benjamin Kaufmann, Andre Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. MIT Press, 2007.

[Gebser *et al.*, 2007c] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczynski. The first answer set programming system competition. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 3–17. Springer, 2007.

[Gelfond and Galloway, 2001] Michael Gelfond and Joel Galloway. Diagnosing dynamic systems in aprolog. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Giunchiglia and Maratea, 2005] Enrico Giunchiglia and Marco Maratea. On the relation between answer set and SAT procedures (or, between smodels and cmod-

els). In *Proceedings of 21st International Conference on Logic Programming (ICLP'05)*, pages 37–51. Springer, 2005.

[Giunchiglia *et al.*, 2004a] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[Giunchiglia *et al.*, 2004b] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-based answer set programming. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 61–66, 2004.

[Giunchiglia *et al.*, 2006] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

[Gomes *et al.*, 1998] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of 15th National Conference on Artificial Intelligence (AAAI)*, page 431437, 1998.

[Gomes *et al.*, 2008] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

[Heljanko and Niemelä, 2003] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3:519–550, 2003.

[Heljanko, 1999] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proceedings Fifth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 218–223, 1999.

[Hietalahti *et al.*, 2000] Maarit Hietalahti, Fabio Massacci, and Nielelä Ilkka. a challenge problem for nonmonotonic reasoning systems. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000.

[Janhunen *et al.*, 2000] T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In *Proceedings 7th Int'l Conf. on Knowledge Representation*, pages 411–419, 2000.

[Janhunen *et al.*, 2006] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Logic*, 7(1):1–37, 2006.

[Koch *et al.*, 2003] Christoph Koch, Nicola Leone, and Gerald Pfeifer. Enhancing disjunctive logic programming systems by sat checkers. *Artificial Intelligence*, 151:177–212, 2003.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 451–465, 2003.

[Lee, 2005] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 503–508. Professional Book Center, 2005.

[Leone and et al., 2005] N. Leone and et al. A disjunctive datalog system dlv (2005-02-23). In *University of Calabria, Vienna University of Technology*, 2005. Available under `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[Li and Anbulagan, 1997] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[Lierler and Maratea, 2004] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 346–350, 2004.

[Lierler, 2008] Yuliya Lierler. Abstract answer set solvers. In *Proceedings of International Conference on Logic Programming (ICLP'08)*, pages 377–391. Springer, 2008.

[Lifschitz and Razborov, 2006] Vladimir Lifschitz and Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7:261–268, 2006.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 1999] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[Lin and Zhao, 2002] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 112–117. MIT Press, 2002.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[Lin *et al.*, 2006] Zhijun Lin, Yuanlin Zhang, and Hector Hernandez. Fast SAT-based answer set solver. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 92–97. MIT Press, 2006.

[Liu and Truszczyński, 2003] Lengning Liu and Mirosław Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. In *Proceedings of CP-2003*, page 495509, 2003.

[Liu and Truszczyński, 2005a] Lengning Liu and Mirosław Truszczyński. Pbmodels – software to compute stable models by pseudoboolean solvers. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2005.

[Liu and Truszczyński, 2005b] Lengning Liu and Mirosław Truszczyński. Properties of programs with monotone and convex constraints. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 701–706, 2005.

[Liu *et al.*, 1998] Xinxin. Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings Fourth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19, 1998.

[Lloyd and Topor, 1984] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[Marek and Subrahmanian, 1989] Victor Marek and V.S. Subrahmanian. The relationship between logic program semantics and non-monotonic reasoning. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proceedings Sixth Int'l Conf.*, pages 600–617, 1989.

[Marek and Truszczyński, 1999] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Marques-Silva and Sakallah, 1996a] João P. Marques-Silva and Karem A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence*, 1996.

[Marques-Silva and Sakallah, 1996b] João P. Marques-Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report, University of Michigan, 1996.

[Mitchell, 2005] David G. Mitchell. A SAT solver primer. In *EATCS Bulletin (The Logic in Computer Science Column)*, volume 85, pages 112–133, 2005.

[Moschovakis, 2008] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008. http://plato.stanford.edu/archives/fall2008/entries/logic-intuitionistic/.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings DAC-01*, 2001.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings Joint Int'l Conf. and Symp. on Logic Programming*, pages 289–303, 1996.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[Nieuwenhuis *et al.*, 2006] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[Nogueira *et al.*, 2001] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 169–183, 2001.

[Nouioua and Nicolas, 2006] Farid Nouioua and Pascal Nicolas. Using answer set programming in an inference-based approach to natural language semantics. In *Proceedings of the Fifth Workshop on Inference in Computational Semantics (ICoS)*, 2006.

[Saccá and Zaniolo, 1990] Domenico Saccá and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 205–217, 1990.

[Simons and Syrjaenen, 2007] P. Simons and T. Syrjaenen. Smodels and lparse – a solver and a grounder for normal logic programs. In *Helsinki University of Technology*, 2007. Available at `http://www.tcs.hut.fi/Software/smodels/`.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Simons, 2000] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, 2000. Adviser-Niemelä, Ilkka.

[Son and Lobo, 2001] Tran Cao Son and Jorge Lobo. Reasoning about policies using logic programs. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Son *et al.*, 2005] Tran Cao Son, Phan Huy Tu, Michael Gelfond, and Ricardo Morales. Conformant planning for domains with constraints-a new approach. In *AAAI*, pages 1211–1216, 2005.

[Stallman and Sussman, 1977] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135196, 1977.

[Syrjanen, 2003] Tomi Syrjanen. Lparse manual[3]. 2003.

[Tang and Ternovska, 2005] Calvin Tang and Eugenia Ternovska. Model checking abstract state machines with answer set programming. In *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. To appear in the Lecture Notes in Computer Science, 2005.

[Tari and Baral, 2005] Luis Tari and Chitta Baral. Using AnsProlog with Link Grammar and WordNet for QA with deep reasoning. In *AAAI Workshop on Inference for Textual Question Answering*, 2005.

[Tseitin, 1968] G.S. Tseitin. On the complexity of derivation in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part II, 1968.

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Walser, 2007] J. Walser. Solving linear pseudo-boolean constraints with local search. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, page 269274, 2007.

[Ward and Schlipf, 2004] J. Ward and J. Schlipf. Answer set programming with clause learning. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, pages 302–313, 2004.

[3] `http://www.tcs.hut.fi/software/smodels/lparse.ps.gz`

[Ward, 2004] Jeffrey Ward. *Answer Set Programming with Clause Learning.* [4]. PhD thesis, 2004. Adviser-Long, Timothy J. and Adviser-Schlipf, Johns S.

[Zhang *et al.*, 2001] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings ICCAD-01*, pages 279–285, 2001.

[4] `http://www.nku.edu/~wardj1/Research/Thesis.pdf`

Vita

Yuliya Lierler (maiden name Babovich) was born in Minsk, Belarus, in 1979, and lived in Minsk until moving to Austin in 1999. She attended the Belarusian State University of Informatics and Radioelectronics from 1996 till 1999. She transferred to the University of Texas at Austin in 1999 where she received the B.S. degree in Computer Sciences in 2000. In 2003, she received the M.S. degree in Computer Sciences at the University of Texas at Austin. She moved to Nuremberg, Germany in 2003 where she worked at the Friedrich-Alexander-Universität Erlangen-Nürnberg for three years. From 2006 onwards, she has been enrolled in the doctoral program in Computer Sciences at the University of Texas at Austin.

Permanent Address: 8111 Davis Mountain Pass
Austin, Texas 78726

This dissertation was typeset with LaTeX 2_ε[5] by the author.

[5] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.