TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Disjunctive Answer Set Programming with Backjumping and Learning

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Mathematik

eingereicht von

Isabella Kammerhofer, BSc.
Matrikelnummer: 01226389

ausgeführt am Institut für Logic and Computation
der Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: **Associate Prof. Marco Maratea**
und **Associate Prof. Dipl.-Ing. Dr.techn. Stefan Woltran**

Wien, am 10.02.2020 Isabella Kammerhofer Stefan Woltran

Kurzfassung

Pseudocode wird in der Literatur häufig verwendet, um Solver oder Algorithmen zu beschreiben und miteinander zu vergleichen. Alternativ dazu kann auch ein so genanntes abstraktes Framework konstruiert werden. Dieses basiert auf Transitionssystemen. Niewenhuis, Oliveras und Tinelli (2006) führten diese Methode ein, um den Davis-Putnam-Logemann-Loveland-Algorithmus, einen Algorithmus zum Erfüllbarkeitsproblem der Aussagenlogik (SAT), zu modellieren und zu analysieren. Die Verwendung abstrakter Frameworks ist eine effektive Methode, mit der bestimmte Eigenschaften wie Endlichkeit, Azyklizität und Korrektheit analysiert, verglichen und bewiesen werden können. Brochenin, Lierler und Maratea (2015) griffen diese Idee auf, um sie auf disjunktives Answer Set Programming anzuwenden. Darüber hinaus konstruierten die Autoren ein generalisiertes Template, das eine Vielzahl möglicher Answer Set Solvers gleichzeitig erfasst. Ihre bisherige Arbeit beschränkte sich jedoch auf disjunktive Answer Set Solver mit Backtracking. Dabei sind die Solver, die sie in ihrer Arbeit erwähnt hatten, mit weiteren Methoden ausgestattet, die in der Arbeit nicht berücksichtigt wurden. Zu diesen Methoden gehören Backjumping, Learning, Forgetting und Restarting. In der vorliegenden Diplomarbeit werden wir einen Schritt weiter gehen und die Arbeit von Brochenin, Lierler und Maratea (2015) um diese weit verbreiteten Techniken erweitern. Wir zeigen, wie ein abstraktes Framework auf diversen, häufig verwendeten Answer Set Solvern aussieht, die die genannten Regeln Backjumping, Learning, Forgetting und Restarting verwenden. Da wir auf die frühere Arbeit aufsetzen, werden wir mit den Solvern CMODELS, GNT und DLV arbeiten, wo wir die zusätzlichen Techniken erfassen. Wir werden auch das generalisierte Template erweitern und uns mit diversen Eigenschaften beschäftigen, die diese Solver mit sich bringen.

Abstract

Pseudocodes are often used in literature to describe and compare solvers or algorithms. Alternatively, we can take the approach of constructing an abstract framework using transition systems. Nieuwenhuis, Oliveras and Tinelli (2006) introduced this method to model and analyse the Davis-Putnam-Logemann-Loveland algorithm for propositional satisfiability. Using abstract frameworks is an effective way to analyse, compare and prove specific features like finiteness, acyclicity and even correctness. Brochenin, Lierler and Maratea (2015) picked up that idea to apply it to disjunctive answer set solvers. Additionally, the authors created a general framework that captures a multitude of possible answer set solvers. The work was restricted to disjunctive answer set solvers with backtracking even though some of the solvers that were being mentioned in their work implement the more general and advanced methods of backjumping and learning, as well as forgetting and restarting. Therefore, we will go a step further and extend their work to these widely used techniques. We illustrate how an abstract framework looks on various commonly used answer set solvers that work with backjumping, learning, forgetting and restarting. Extending the earlier work, we will work with the solvers CMODELS, GNT and DLV, where we will capture backjumping, learning, forgetting and restarting. We will also extend the general template from the earlier work and observe the earlier mentioned features from a more general point of view.

Acknowledgement

At this point, I would like to express my gratitude to all people that have supported me throughout my studies and in particular during the work on my thesis.

First of all, I want to thank my advisor Marco Maratea for his continuous advice, support and insight througout my work on the thesis. He introduced me to this topic during a seminar that he held on solving algorithms in terms of Abstract Solvers for Propositional Satisfiability, while guest lecturing in Vienna. This led to an ERASMUS+ semester abroad getting started on the thesis in Genova. Thanks for the many skype meetings and meetings in person even though it wasn't always easy due to the local distance. Thanks for giving me the chance to work on an interesting topic, that can get quite challenging to think about. It was my pleasure to work with you.

Also, I am thankful to Stefan Woltran, my official supervisor, for the support, the quick responses and the opportunity of going to Italy to work with Marco. Thanks for the valuable inputs and remarks on the manuscript!

Ein besonderer Dank gilt meinen Eltern. Danke für all die Liebe und Unterstützung, die ihr mir entgegenbringt. Danke dafür, dass ich stets meine eigenen Entscheidungen treffen durfte und ihr mir stets den Rücken gestärkt habt. Danke, dass ihr immer für mich da seid.

Thanks to all my friends for making the years of studying such a great time that I will always treasure.

And last, but not least, thank you Marcel, for being in my life. Thanks for being such a wonderful, easy-going person. I'm looking forward to many more years with you.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am February 16, 2020

Isabella Kammerhofer

Contents

1 Introduction

Context In declarative programming we tell a program **what** needs to be done unlike to imperative programming, where we answer the question of **how** to do something. Answer set programming (ASP) is a form of declarative programming. It is oriented towards difficult combinatorical search problems, see [MT99], [Nie99] and [Lif08]. A combinatorical search problem is an optimization problem that is easy to state and has a finite, but often large number of feasible solutions.

In ASP, we represent given problems by using so-called logic programs. A logic program can be seen as a list of rules and facts that we know about the problem. These programs have a similar syntax as programs in Prolog, which is another logic programming language, but one of the main differences is the output.

In the following example, we see two facts: Mary is a woman. And Mary is a parent. The third line is a rule expressing that for Mary being a mother, Mary has to be a woman and Mary has to be a parent. We call $mother(mary)$ the head of the rule and $woman(mary), parent(mary)$ the body.

$$\begin{aligned} woman(mary).& \\ parent(mary).& \\ mother(mary) \leftarrow woman(mary), parent(mary).& \end{aligned} \tag{1.1}$$

The solution of such a program in ASP is a so-called answer set, see [Lif99], or stable model, according to [GL88]. A program that calculates such answer sets is called answer set solver. A search procedure they are typically based on is the boolean satisfiability problem (SAT), where we are interested in the question whether a formula is satisfiable. For a reference see [Lie10].

ASP has a broad range of applications in research and practice, for example in robotics, see [EAP12], scheduling like in [DM17], or space shuttle control as described in [NBG+01]. But as there are many applications, answer set programming also has more theoretical components to explore.

State of the Art The most common tool to describe and compare computation procedures is pseudocode. A representation by pseudocode in ASP was used for example in [GM05] and [GLM08]. Over the years, other formal approaches were created, like the one from Gebser and Schnaub, using tableau calculi, see [GS06] or [GS13], and the one that we will be using in this thesis, the use of abstract frameworks via transition systems that is based on [NOT06]. There, the authors described the Davis-Putnam-Logemann-Loveland (DPLL, see [DP60] and [DLL62]) procedure using a graph, where the nodes are states of computation and the edges are the allowed transitions between the states. The DPLL-

algorithm can be used to solve the SAT problem. Executing the procedure leads to a path in the graph. By ignoring heuristics and implementation issues, the abstract approach can simplify comparison, understanding and proving features like completeness or termination in an easier way.

This idea was taken up in [Lie11] and [LT11] to describe answer set solvers for non-disjunctive programs such as SMODELS and CMODELS. Non-disjunctive refers to the form of the rules of the program. The rule in Equation 1.1 is a non-disjunctive rule, as there is only one term in the head of the rule. In the rule below we express by a disjunctive rule that when we know that Alex is a parent then he is either a father or she is a mother.

$$mother(alex), father(alex) \leftarrow parent(alex). \tag{1.2}$$

Later, an abstract framework was being created for disjunctive answer set solvers, such as CMODELS, GNT and DLV in [BLM14] and [BLM16]. For more information on CMODELS we refer to [Lie05], for GNT, see [JNS+06] and DLV was being introduced in [LFP+06].

The authors even created a general framework, a graph template that works for all the mentioned disjunctive answer set solvers. However, major techniques like backjumping and learning, critical for solver performance, were not taken into account.

Contribution In this thesis, we want to fill this gap by extending the existing work of capturing disjunctive answer set solvers like CMODELS, GNT and DLV in an abstract framework with backjumping and learning.

We construct new graphs that capture backjumping and learning, as well as forgetting and restarting. Backjumping is a form of backtracking that can improve efficiency by changing a decision that is the reason for an inconsistency, which is an erroneous state, without having to consider in what order the decisions were made. Applying backjumping can lead to clauses that are worth remembering in the future, which is done by learning. These learned clauses can be removed as well, in our terms "forgotten". Restart is being used to do a reset to the starting point while keeping the learned clauses, which could lead to a more efficient way of finding a solution.

We use the abstract framework to represent various answer set solvers that can handle disjunctive programs and prove correctness and termination, and look at properties like finiteness and acyclicity. In the earlier work, technically multi-sets are being used for representing critical components of abstract frameworks. We will see, that by using sets instead of multi-sets, we will obtain acyclicity in some cases as additional feature.

After having a look at the specific answer set solvers, we extend the graph template presented in [BLM16] to backjumping and learning as well as forgetting and restarting. A graph template is a general framework, that accounts for major techniques used in disjunctive answer set solvers. Again, we will take a look at finiteness, acyclicity, termination and completion. This graph template can be used to create new abstract answer set solvers

that work on disjunctive programs and can handle backjumping, learning, forgetting and restarting.

Structure of the document The content of this thesis is structured as follows.
Chapter 2 introduces the abstract framework by revewing abstract DPLL using CNF formulae. We present the preliminaries needed to discuss that topic, where the main references come from [NOT06], while the notation coincides with [BLM16]. This chapter provides a first understanding of the rules of backjumping and learning that we want to transfer to disjunctive programs.

Chapter 3 is an introduction to disjunctive answer set programming, where we deal with the basics of answer sets programming which we will need to understand the further work.

Chapter 4 reviews the work of Brochenin, Lierler and Maratea, where they presented an abstract solver for three important answer set solvers, namely CMODELS, GNT and DLV, that can handle disjunctive programs. This chapter is limited to the status quo - answer set solvers without backjumping and learning, that use backtracking to unravel conflicts.

The previous chapters provide the tools for the extension of the solvers to backjumping and learning in Chapter 5. This chapter gives an overview of the changes made to the previously done work to receive a system that captures backjumping and learning on three main solvers, CMODELS, GNT and DLV. Motivated by the results of [BLM16] we determine whether some features like acyclicity, termination and correctness hold in the extended versions. We also take a look at the extension to some other rules that were introduced in Chapter 3, namely forgetting and restarting.

In Chapter 6 we discuss another main result of [BLM16] where they introduced an abstract framework that captures the solvers in the previous chapter and elaborate the extension to backjumping, learning, forgetting and restart as well.

In Chapter 7, we can find a small conclusion, summarizing the main contributions of the thesis.

2 Abstract DPLL

A common access point to get into the topic of abstract answer set programming is reviewing the abstract DPLL framework. In 1962 the authors introduced the Davis-Putnam-Logemann-Loveland (DPLL) procedure ([DP60], [DLL62]). DPLL can be used to decide the satisfiability of propositional logic formulas in concunctive normal form. This is achieved by exploring sets of literals to generating classical models of that formula. The procedure is similar to the algorithms used in efficient boolean satisfiability problem (SAT) solvers. For a reference see [GKSS08].

In 2006, the authors R. Nieuwenhuis, A. Oliveras and C. Tinelli picked up the procedure and developed an "abstract" way of describing SAT solvers. This abstract approach can be extended to ASP solvers that use these features. Abstract DPLL is a transition-rule-based formulation of DPLL to simplify the understanding of properties like completeness or termination by neglecting heuristics and implementation issues and therefore getting a more abstract point of view.

Additionally, several enhancements of DPLL were made in the last years, for example non-chronological backtracking, also known as backjumping, conflict-driven lemma learning and restarts.

In this chapter we want to get an overview of the original rules that were extended over the years.

2.1 Preliminaries of SAT and propositional logic in general

In the following chapter, most definitions and notations were taken from [BLM16].

Definition 1. An *atom* a denotes a Boolean variable, i.e.

$$a \in \{true, false\},$$

which can be written as

$$a \in \{\top, \bot\},$$

respectively.

Definition 2. A *literal* l is either an atom a or its negation $\neg a$. $\bar{l}$ is the complement of a literal l, i.e. $\neg a$ for a literal a and a for a literal $\neg a$.

Definition 3. A *conjunction* $\wedge$ is the logical AND-function. A *disjunction* $\vee$ is the logical OR-function.

Figure 2.1 and Figure 2.2 show the outcome of the application of the conjunction and disjunction operator on two boolean values. We see that $A \wedge B$ is true only if both A and B are true. In contrary, $A \vee B$ is false only if both A and B are false.

$\wedge$	true	false
true	true	false
false	false	false

Figure 2.1: truth table of the conjunction operator

$\vee$	true	false
true	true	true
false	true	false

Figure 2.2: truth table of the disjunction operator

Definition 4. *[BLM16]* For a conjunction (respectively a disjunction) of literals D, by $\overline{D}$ we denote the disjunction (respectively the conjunction) of the complements of the elements of D.

Example 1. Let D be $(\neg a \vee b) \wedge c$. Then, $\overline{D}$ equals $(a \wedge \neg b) \vee \neg c$.

Definition 5. A *clause* C is a finite disjunction of literals. We identify an empty clause with the symbol $\bot$.

Example 2. Examples of clauses include $a \vee \neg b$, $\bot$, b, $c \vee d$.

Remark 1. A conjunction (respectively a disjunction) of literals can be viewed as a set, containing each of the conjunction's (disjunction's) literals. As a clause is a disjunction of literals, it can be identified as a set as well. Therefore, there are no repetitions of literals in a clause.

Example 3. Let $C = a \vee b \vee \neg c \vee \neg d$ be a conjunction of literals. The according set is $\{a, b, \neg c, \neg d\}$.

Definition 6. *[BLM16]* A *CNF formula* is a finite conjunction (alternatively, a set) of clauses, where CNF stands for conjunctive normal form. Since a CNF formula is identified with a set of clauses, there are no repetitions of clauses in a CNF formula.

Example 4. An example of an expression in conjunctive normal form is $a \wedge (c \vee \neg d)$, which is equal to the set $\{a, c \vee \neg d\}$.

Definition 7. Let L be a set of literals. The disjunction of the elements of L can be written as $L^{\vee}$, while the conjunction can be written as $L^{\wedge}$. We define $atoms(L)$ as the set of atoms occurring in L.

Example 5. Let $L = \{a, \neg b, c\}$. Then $L^{\vee} = a \vee \neg b \vee c$, $L^{\wedge} = a \wedge \neg b \wedge c$ and $atoms(L) = \{a, b, c\}$.

Definition 8. Let L be a set of literals. The set L^{+} contains all atoms that occur positively in L, the set L^{-} contains all atoms that occur negatively in L.

Example 6. Let $L = \{a, \neg b, c\}$. Then $L^{+} = \{a, c\}$ and $L^{-} = \{b\}$.

Definition 9. Let X be a set of atoms and L be a set of literals. By using $L_{|X}$ we mean the maximal subset of L over the set X.

Example 7. $\{a, b, \neg c\}_{|\{b,c\}} = \{b, \neg c\}$.

Definition 10. A *(total truth) assignment* to a set X of atoms is a function

$$f : X \to \{false, true\}.$$

Definition 11. An assignment M *satisfies* a formula F, if F evaluates to *true* under this assignment. An assignment that satisfies a formula F is called a *satisfying assignment* or a *(classical) model* for F. We also write

$$M \models F.$$

Reversely, an assignment *contradicts* a formula F, if F evaluates to *false* under an assignment. If all assignments of a formula F contradict that formula, the formula F has no model and we call F *unsatisfiable*.

Definition 12. A set L of literals is *consistent*, if it does not contain both, a literal and its complement.

Let X, Y be sets of atoms such that $X \subseteq Y$. We identify X with an assignment to Y, if

$$a \mapsto \begin{cases} true, & a \in X \\ false, & a \in Y \backslash X \end{cases}$$

for all atoms $a \in X \vee Y$.
Now let L be a consistent set of literals. We can identify L with an assignment to $atoms(L)$, if

$$a \mapsto \begin{cases} true, & a \in L \\ false, & \neg a \in L \end{cases}$$

for all atoms $a \in L$.

Definition 13. A set M of literals is *complete* over the set of atoms X if $atoms(M) = X$.

Remark 2. A complete and consistent set of literals over X represents an assignment to X.

Definition 14. Let F, F' be formulas. We say that F' is a *logical consequence* of F, if each model in F is a model for F' as well. If $F \models F'$ and $F' \models F$, we say that F and F' are *logically equivalent*.

2.2 Classical abstract DPLL

In abstract DPLL, so called transition systems are being used instead of pseudocode, which makes it easier to prove features like correctness, compare different solvers or even to design new solvers. In [NOT06], the authors introduced an abstract framework using directed graphs, where every execution of the DPLL procedure corresponds to a path in the graph. The nodes of the graph will be described by states. To describe the edges of the graph we will use a transition system.

Definition 15. *[Lie11]* A *record* M relative to a set of atoms X is a list of literals over X, where

- some literals in M can be so-called decision literals, which are being marked by Δ, and
- M contains no repetitions.

Note that a record is actually an assignment, where the decision literals can be seen as literals, i.e.

$$l^\Delta = l \text{ and } \neg l^\Delta = \neg l.$$

Additionally, we say that a literal l is unassigned by a record if it contains neither l nor $\neg l$.

A record can also be seen as a set containing all the elements of the record disregarding their annotations. For instance, the record $\neg a b^\Delta c$ can be identified with the set $\{\neg a, b, c\}$.

Definition 16. *[BLM16]* A *state* relative to a set of atoms X is either a record relative to X, the state $Ok(L)$, where L is a record relative to X or the distinguished state $Failstate$.

Example 8. *[Lie11]* The states relative to a singleton set $\{a\}$ of atoms are

$$\begin{gathered} Failstate,\ \emptyset,\ a,\ \neg a,\ a^\Delta,\ \neg a^\Delta,\ a\neg a,\ a\neg a^\Delta \\ a^\Delta\neg a,\ a^\Delta\neg a^\Delta,\ \neg aa,\ \neg aa^\Delta,\ \neg a^\Delta a,\ \neg a^\Delta a^\Delta, Ok(a). \end{gathered} \tag{2.1}$$

By $\emptyset$ we denote the empty list.

Remark 3. Note that in Example 8, only $\emptyset$, a, $\neg a$, a^Δ and $\neg a^\Delta$ are consistent, as they are the only states that do not contain both the literal and its complement.

Definition 17. *[Lie11]* A *transition rule* is an expression of the form $M \Longrightarrow M'$ followed by a condition, where M and M' are nodes of the constructed graph.

Definition 18. [JNS+06] A *step* is an application of a transition rule.

Definition 19. *[NOT06]* A *transition system* is a set of transition rules defined over some given set of states.

Definition 20. *[NOT06]* Let S and $(S_i)_{i=0,\ldots,n}$ be states. A *transition relation* $S \Rightarrow S'$ is a binary relation that represents the transition from one state S to another state S'.

2.2.1 The graph DP_F

Now we want to define a graph that represents the application of DPLL. Therefore, let F be an arbitrary CNF formula. By applying the DPLL procedure to F, we can construct the graph DP_F. The set of nodes are the states relative to the set of atoms of F.

Note that the states $Failstate$ and $Ok(L)$ where L is a set of literals, due to Definition 16. In other words, a state in the DPLL graph is a hypothetical state of the DPLL computation. We say, that a node in the graph is *terminal*, if no edge originates from it. The

state $\emptyset$ is the *initial* state. The edges of the graph DP_F are defined by the transition rules presented in Figure 2.3.

Here, the transition rules of DPLL are *Conclude*, *Backtrack*, *Unit*, *Decide* und *Success*. There are different variations for these transition rules, e.g. in some papers the rule *Success* is not being explicitly mentioned.

By applying the rule *Unit* we can add a literal that is the logical consequence of the given formula and the previous decisions. The rule *Decide* lets us assign an arbitrary value to an atom. By applying it, we add a new (decision) literal to the record. When we assign an arbitrary value to the atom using *Decide*, we could find out at some point, that we made the wrong choice earlier. In that case, we can apply the rule *Backtrack*. Here we can fix the last decision that we made at an earlier point, by changing the decision literal to its complement. The rule *Success* is an indicator that the current state of computation is a model of our program. If the formula is not satisfiable, the rule *Conclude* will be applied.

$$
\begin{array}{lll}
\mathit{Conclude}: & L \Longrightarrow \mathit{Failstate} & \text{if } \left\{ \begin{array}{l} L \text{ is inconsistent and} \\ L \text{ contains no decision literals} \end{array} \right. \\[2ex]
\mathit{Backtrack}: & Ll^{\Delta}L' \Longrightarrow L\bar{l} & \text{if } \left\{ \begin{array}{l} Ll^{\Delta}L' \text{ is inconsistent and} \\ L' \text{ contains no decision literals} \end{array} \right. \\[2ex]
\mathit{Unit}: & L \Longrightarrow Ll & \text{if } \left\{ \begin{array}{l} l \text{ does not occur in } L \text{ and} \\ F \text{ contains a clause } C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{array} \right. \\[2ex]
\mathit{Decide}: & L \Longrightarrow Ll^{\Delta} & \text{if } \left\{ \begin{array}{l} L \text{ is consistent and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } L \end{array} \right. \\[2ex]
\mathit{Success}: & L \Longrightarrow \mathit{Ok}(L) & \text{if no other rule applies}
\end{array}
$$

Figure 2.3: Transition rules of classical DPLL, edges of the graph DP_F.

Remark 4. By constructing the graph, we see, that a path in the resulting graph describes the process of the search for a classical model of a formula. This is being visualized in Example 9.

Example 9. Let F be represented by the set $\{a \vee \neg b, \neg a \vee c, \neg a \vee \neg c\}$. One of the resulting graphs is being shown in Figure 2.4.

The initial state is $\emptyset$. In a first step we apply the rule *Decide*. We can do this, because $\emptyset$ is consistent and the atom a does not occur in our assignment. In a next step we apply the rule *Unit*, because adding the literal c to our assignment is a logical consequence of our previous decision and the given formula. The literal c does not occur in our assignment yet and the second clause $\neg a \vee c$ can only be satisfied, if we assign the value true to the atom c, because we already mapped the atom a to true. The same statement holds for the third clause of the formula and $\neg c$. But now the assignment is inconsistent. Therefore either the rule *Conclude* or *Backtrack* hold. As there is a decision literal in our assignment, we

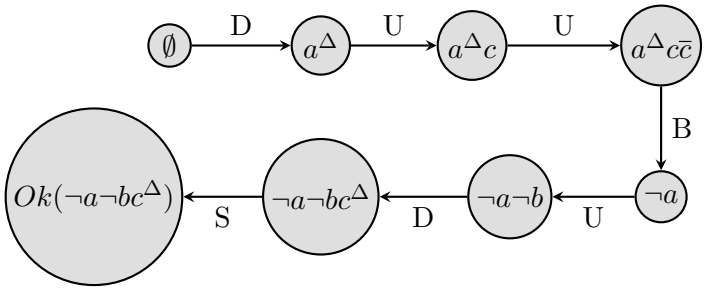

Figure 2.4: A possible graph of F

must apply *Backtrack* and therefore change the value of the atom a to false. Now the rule *Unit* holds for the first statement, where we have to assign a value to the atom b. Then, we could choose either c or $\neg c$, so we write c^Δ, using the *Decide* rule. We can see that no more rule applies. Therefore we need to apply *Success*.

Using similar arguments, the path of the graph could also look like in Figure 2.5 or 2.6.

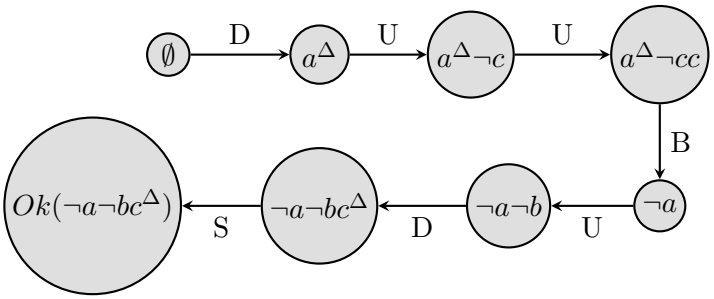

Figure 2.5: Alternative path: permute the order of c and $\neg c$

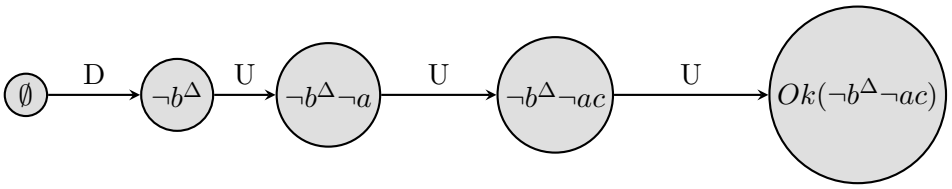

Figure 2.6: Alternative path: without backtracking

An alternative way to describe the path of the graph DP_F is to use transition relations. Additionally, we write down the transition rule that was being used after each $\Rightarrow$ to justify the presence of the edge in the graph. In the path of Equation 2.2 we can see the according transcription of Figure 2.4.

$$\begin{array}{lcll} \emptyset & \Rightarrow & (Decide) & \\ a^{\Delta} & \Rightarrow & (Decide) & \\ a^{\Delta}c^{\Delta} & \Rightarrow & (Unit) & \\ a^{\Delta}c^{\Delta}\neg c & \Rightarrow & (Backtrack) & (2.2) \\ \neg a & \Rightarrow & (Unit) & \\ \neg a\neg b & \Rightarrow & (Decide) & \\ \neg a\neg bc^{\Delta} & \Rightarrow & (Success) & \end{array}$$

Definition 21. A graph is *acyclic* when there are no cycles, see [BM76, p. 25]. This means that when we follow a graph from node to node, you will never visit the same node twice.

Proposition 1. *[Lie11] For any CNF formula F*

1. *graph DP_F is finite and acyclic,*
2. *any terminal state of DP_F other than $Failstate$ is $Ok(L)$, where L is a model of F,*
3. *$Failstate$ is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable.*

Proof. For the finiteness consider a state S of the graph DP_F. S can either be a record relative to $atoms(F)$, the state $Ok(L)$ or the state $Failstate$. Since F is a CNF formula, it is finite and therefore consists of finitely many atoms. These finitely many atoms plus the special states $Failstate$ and $Ok(L)$ lead to a finite amount of states.

For a proof of the other properties see [Lie11, Proposition 1]. □

2.2.2 The graph DPL_F

In practice, it may not be enough to implement the classical DPLL system as described in Section 2.2.1, because of efficiency reasons. For example, if the CNF formula has many atoms and the assignment contains many decision literals, there could be the case that the rule *Backtrack* needs to be applied multiple times until we find the root of the inconsistency, as we can only undo the latest decision. Most DPLL-based SAT solvers use a more general and powerful form of backtracking, which is called *backjumping*. While backtracking always undoes the last decision being made, backjumping allows to skip decisions that are not the reason of the inconsistency and change a decision literal that is part of the inconsistent term. Therefore it is possible to undo several decisions at once and backtrack further in the search tree.

In [Lie11, section 6] the graph DP_F is being extended by adapting and adding some rules of classical $DPLL$. The idea comes from [NOT06, section 2.4]. To do this, we need a new form of state, an augmented state.

Definition 22. Let F be a CNF formula. An *augmented state* to F can either be a pair $M\|\Gamma$ with M being a record relative to $atoms(F)$ and Γ being a set of clauses over atoms of F that are entailed by F, the state $Ok(L)$ or the distinguished state $Failstate$.

Note that Γ is a set and not a multi-set like in the previous papers as this will give us the chance to capture graphs that are acyclic.

One of the transition rules that will be added is Backjumping. Backtracking is a special case of backjumping, where the decision literal that is being changed, is the last decision that was made.

We can define a new graph that is basically the graph DP_F, but the transition rule *Backtrack* is being overwritten by the new rule *Backjump* in Figure 2.7.

$$\textit{Backjump}: \qquad Ll^{\Delta}L' \| \Gamma \Longrightarrow L\bar{l} \| \Gamma \quad \text{if} \begin{cases} Ll^{\Delta}L' \text{ is inconsistent} & \text{and} \\ F \models \bar{l} \vee \overline{L} \end{cases}$$

Figure 2.7: Backjump rule

Example 10. Let's consider the same CNF formula F as in Example 9. We recall, the according set $\{a \vee \neg b, \neg a \vee c, \neg a \vee \neg c\}$. One path of the graph DP_F with the rule *Backtrack* could be the following:

$$\begin{array}{lcl} \emptyset & \Rightarrow & (\textit{Decide}) \\ a^{\Delta} & \Rightarrow & (\textit{Decide}) \\ a^{\Delta}b^{\Delta} & \Rightarrow & (\textit{Unit}) \\ a^{\Delta}b^{\Delta}c & \Rightarrow & (\textit{Backtrack}) \\ a^{\Delta}\neg b & \Rightarrow & (\textit{Unit}) \\ a^{\Delta}\neg bc & \Rightarrow & (\textit{Backtrack}) \\ \neg a & \Rightarrow & (\textit{Unit}) \\ \neg a\neg b & \Rightarrow & (\textit{Decide}) \\ \neg a\neg bc^{\Delta} & \Rightarrow & (\textit{Success}) \end{array} \tag{2.3}$$

We see, that we need to apply the rule *Backtrack* twice, as we can only undo the latest decision. When a is assigned to *true* and c is assigned to *true*, then the third clause $\neg a \vee \neg c$ is not fulfilled. Therefore, we need to apply *Backtrack*. We see that b does not have anything to do with the reason for inconsistency.

When changing the *Backtrack* rule to the *Backjump*, we can skip changing value of the atom b. Therefore in the graph similar to DP_F where we replace backtracking by backjumping, the path could look as following:

$$\begin{array}{lcl} \emptyset & \Rightarrow & (\textit{Decide}) \\ a^{\Delta} & \Rightarrow & (\textit{Decide}) \\ a^{\Delta}b^{\Delta} & \Rightarrow & (\textit{Unit}) \\ a^{\Delta}b^{\Delta}c & \Rightarrow & (\textit{Backjump}) \\ \neg a & \Rightarrow & (\textit{Unit}) \\ \neg a\neg b & \Rightarrow & (\textit{Decide}) \\ \neg a\neg bc^{\Delta} & \Rightarrow & (\textit{Success}) \end{array} \tag{2.4}$$

We know where the inconsistency lies and we skip the change of the assignment of the atom b and can fix the value of the atom, where it is actually needed in one step.

Whenever there is a conflict, where *Backjump* needs to be applied, there is always a clause that is the reason for the inconsistency, according to [NOT06], where the authors call such a clause a *backjump clause*.

In Example 10 the backjump clause is $\neg a \vee \neg c$, which we did not consider when assigning c to *true* in Equation 2.4.

Remembering this backjumping clause will lead to *conflict-driven learning*. This is where the set Γ from the augmented state comes into play. It is being used to capture newly learned clauses without changing the original formula.

We add the possibility of learning additional clauses. The motivation to remember such new clauses, is to prevent future similar conflicts from happening. We do this by adding some clauses to a set of *learned clauses*.

Most modern SAT solvers use both, backjumping and conflict-driven learning, to receive a more efficient way finding of a solution. The learned clause is being saved to the clause database and used for further decisions, which transition rule should be applied. In this framework, we describe backjumping in general, but we do not go into depth on how a solver finds the backjump clauses, that can be learned.

As we can capture learned clauses, we need a transition rule that can add these to Γ, which can be found in Figure 2.8.

$$\textit{Learn}: \qquad M\|\Gamma \Longrightarrow M\|C,\Gamma \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ and} \\ F \models C \end{cases}$$

Figure 2.8: Learn rule

To use the learned clauses, we need to adapt the rule Unit Propagate accordingly, as we want to consider the clauses in the CNF formula, as well as the learned clauses.

$$\textit{UnitDPL}: \qquad M\|\Gamma \Longrightarrow Ml\|\Gamma \quad \text{if} \begin{cases} C \vee l \in F \cup \Gamma \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

Figure 2.9: Unit Propagate rule in DPL_F

We call the new graph DPL_F. Its nodes are the augmented states relative to F. Its transition rules *Decide*, *Conclude* and *Success* of DP_F are being extended to

$$\begin{array}{lcll} (M\|\Gamma \Longrightarrow M'\|\Gamma) & \Leftrightarrow & (M \Longrightarrow M') & \text{and} \\ (M\|\Gamma \Longrightarrow \mathit{Failstate}) & \Leftrightarrow & (M \Longrightarrow \mathit{Failstate}) & \text{and} \\ (M\|\Gamma \Longrightarrow \mathit{Ok}(M)) & \Leftrightarrow & (M \Longrightarrow \mathit{Ok}(M)). & \end{array} \tag{2.5}$$

On the left we see the new form of the transition rule, on the right is the old form of the transition rule.

We see, that the rules stay basically the same, but the set of learned clauses is being remembered when applying *Decide*. Besides the *Decide*, *Conclude* and *Success* rule from Figure 2.3, that are being extended by using Equation 2.5, the transition rules of DPL_F can be found in Figure 2.7, 2.8 and 2.9.

From now on, we will omit the word "augmented" before "state" when it is clear from the context.

Theoretically, the rule *Learn* is always applicable, so we can't say that a state is terminal. Therefore, we introduce semi-terminal nodes.

Definition 23. A node in the graph is *semi-terminal* if no rule other than *Learn* is applicable to it.

The graph DPL_F can be used for deciding the satisfiability of a formula F by constructing an arbitrary path from the initial state $\emptyset\|\emptyset$ to a semi-terminal node.

The following proposition describes the features of the graph DPL_F.

Proposition 2. *[Lie11] For any CNF formula F,*

1. *every path in DPL_F contains only finitely many edges justified by the transition rules Unit, Backjump, Decide and Conclude.*
2. *for any semi-terminal state $M\|\Gamma$ of DPL_F reachable from $\emptyset\|\emptyset$, M is a model of F,*
3. *Failstate is reachable from $\emptyset\|\emptyset$ in DPL_F if and only if F is unsatisfiable.*

Proof. See [Lie11, Proposition 7] □

Example 11. *[Lie11]* Let F be the formula

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

One path in DPL_F can be the following:

$$\begin{aligned}
\emptyset\|\emptyset &\Rightarrow (Learn)\\
\emptyset\|b \vee c &\Rightarrow (Decide)\\
\neg b^{\Delta}\|b \vee c &\Rightarrow (UnitDPL)\\
\neg b^{\Delta} c\|b \vee c &\Rightarrow (UnitDPL)\\
\neg b^{\Delta} ca\|b \vee c &\Rightarrow (Success)\\
Ok(\neg b^{\Delta} ca)
\end{aligned}$$

If we started with the literal a we have to apply $UnitDPL$ and add c. Starting with the literal $\neg a$ would lead to the literal b on applying $UnitDPL$. Therefore, a rule that could be learned in a first step is $b \vee c$. It will be considered in the further steps.

So when assigning the value $false$ to b after learning the rule $b \vee c$, we can apply $UnitDPL$, which would not be possible by just using the rules of the formula. Afterwards we'd have to assign a to $true$. Otherwise, the first disjunction would not be satisfied.

In total, the application of the transition rules of DPL_F will lead to a semi-terminal state $\{\neg b, c, a\}$, which is a model of F by Proposition 2 (b).

2.2.3 Further Extensions of the graph DPL_F

Modern SAT solvers often implement techniques as forgetting and restart in addition to backjumping and learning.

Forget
As we can learn many different clauses, when applying *Learn*, there is a possibility that the CNF formula including its learned clauses turns out to be very big, which could have a negative effect on the performance of the solver. Therefore it can be helpful to introduce a new transition rule: *Forget*. It is the opponent of the transition rule *Learn*, where learned clauses can be removed when the conflicts are not very likely to be found again. When a solver "recognizes" that a clause, that was being learned earlier is not helpful anymore, that clause is being deleted from the set of learned clauses. The transition rule in Figure 2.10 describes this process of forgetting a clause.

$$Forget: \qquad M \| \Gamma, C \Longrightarrow M \| \Gamma \quad \text{if } \{ \ F \models C$$

Figure 2.10: Forget rule

Restart
In the case that the search is not making "enough" progress according to some measure, the transition rule *Restart* was being introduced. The idea is that the additionally learned clauses will lead the heuristics for the rule *Decide* to behave differently, so that the search space is being explored in a more compact way. This is done by starting the search from scratch in addition to keeping the clauses that were being learned up to this point. The motivation behind this, is that the additional knowledge that is gained by the learned clauses will lead the heuristics to behave differently and more efficiently.

$$Restart: \qquad M \| \Gamma \Longrightarrow \emptyset \| \Gamma$$

Figure 2.11: Restart rule

Remark 5. Keep in mind that restarting too early and too often could eventually lead to the consequence that the program will not terminate. We could also lose completeness due to the early termination of exploring the search space before we start over using the rule *Restart*. To ensure in practice that a final state is being reached at some point, the minimal number of steps is being increased between each pair of restart steps.

Remark 6. The transition rules *Restart* and *Forget* are similar to *Learn* in a sense that they can be applied at any time. Therefore the definition of a semi-terminal state has to be changed: A node in the graph is *semi-terminal* if no rule other than *Learn*, *Forget* and *Restart* is applicable to it.

Proposition 2 holds for both, adding the transition rule *Forget* and *Restart*. For the first part, the new rules do not change the finiteness, as they are not being considered. The second part is still valid, as both *Restart* and *Forget* are being treated like *Learn*, so they do not lead to a semi-terminal state. Neither *Forget* nor *Restart* lead to the state *Failstate*, so the third part of the proposition is independent of *Forget* and *Restart* as well.

Remark 7. To keep finiteness as well as completeness, while including the transition rules *Learn*, *Forget* and *Restart*, we need to add some constraints to our formulation. To achieve this, we have to look at subderivations and periodicities.

Definition 24. *[NOT06]* A *derivation* is a sequence of transitions of the form $S_0 \Rightarrow S_1, S_1 \Rightarrow S_2, \ldots$, denoted by the following:

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \cdots$$

A *subderivation* is a subsequence of a derivation.

Definition 25. *[NOT06]* *Restart* has *increasing periodicity* in a derivation, if for each subderivation $S_i \Rightarrow \cdots \Rightarrow S_j \Rightarrow \cdots \Rightarrow S_k$ with S_i, S_j and S_k being *Restart* steps, the number of transitions in $S_i \Rightarrow \cdots \Rightarrow S_j$ is strictly smaller than in $S_j \Rightarrow \cdots \Rightarrow S_k$.

Proposition 3. *The graph DPL_F that is being extended by the rules Forget and Restart is finite if it contains no infinite subderivations consisting of only Learn and Forget steps and if Restart has increasing periodicity in it.*

Proof. The proof follows [NOT06, Theorem 2.16] □

Remark 8. Proposition 2 holds for the extended graphs DPL_F with forgetting as well as the extended graph DPL_F with forgetting and restarting, as the first and the second part excludes learning, forgetting and restarting. Neither of the rules *Forget* and *Restart* lead to *Failstate*, therefore the third part should follow the lines of [Lie11] as well.

3 Disjunctive ASP

Answer set programming (ASP) is a form of declarative programming commonly used for solving difficult, primarily NP-hard search problems, according to [Lif08]. It consists of rules that look like the rules in Prolog[1], but the computational mechanism that is used in ASP is different. In ASP these are based on fast satisfiability solvers for propositional logic.

In ASP, search problems are reduced to computing stable models (answer sets). In practice, some of the answer set solvers are enhancements of the DPLL procedure. ASP solvers like CMODELS follow the so called SAT-based approach where a SAT solver is invoked for search. Other ASP solvers are being influenced by SAT solvers by adopting computational techniques from them. For instance, the solver DLV implements backjumping (cf. [RFL06]).

An early application of answer set programming was being introduced in 1997, where the authors introduced a planning method, see [DNK97]. Some applications of answer set programming are Configuration of systems, e.g. the Partner Units Problem for configuring parts of a railway safety system of Siemens [Tep17], Planning and Scheduling, e.g. nurse scheduling [DM17], Classification or Bioinformatics, to name a view. For more information see [FFea18].

Here, we will work with disjunctive logic programs. The problem of deciding whether it has an answer set is $\sum_2^P$-complete, because there is an exponential number of possible candidate models and the hardness of checking whether a candidate model is an answer set of a program is co-NP-complete (cf. [BLM14]). Because of the high complexity, only a few solvers can handle this type of program. In the next chapters we will take a closer look at some of these solvers, namely DLV, CMODELS and GNT.

In this chapter, most definitions and notations are taken from [BLM16].

3.1 Disjunctive Logic Programs

Definition 26. A *rule* is a construct of the form

$$A \leftarrow B, \tag{3.1}$$

where A is the head and B is the body. If A is empty, it is being dropped from the expression.

Definition 27. The *head* is a finite disjunction

$$A = a_1 \vee a_2 \vee ... \vee a_n, \tag{3.2}$$

[1] The logical programming language Prolog, whose name comes from "programmation en logique", is also a common part declarative programming.

where each a_i $(1 \leq i \leq n)$ is either an atom or the symbol $\bot$. If A consists of more than one atom, the rule is called disjunctive. A non-disjunctive rule has at most one single atom in the head.

Definition 28. The *body* is of the form

$$b_1, ..., b_l, \text{not } b_{l+1}, ..., \text{not } b_m, \tag{3.3}$$

where each b_i ($1 \leq i \leq m$) is an atom. We call $b_1, ..., b_l$ the positive and not $b_{l+1}, ..., \text{not } b_m$ the negative part of the body, and can write B^+ and B^- respectively.

Definition 29. A non-disjunctive rule, where the body B is empty, is called a fact. The syntax for a fact is

$$a_1.$$

Remark 9. We can view Equation 3.3 as the conjunction

$$b_1 \wedge ... \wedge b_l \wedge \text{not } b_{l+1} \wedge ... \wedge \text{not } b_m. \tag{3.4}$$

Further, we can identify a rule with the clause

$$A \cup \overline{B} = a_1 \vee ... \vee a_n \vee \overline{b_1} \vee ... \vee \overline{b_l}, b_{l+1} \vee ... \vee b_m. \tag{3.5}$$

Conversely, we can identify CNF formulas with logic programs.

We say, that a rule $A \leftarrow B$ is equivalent to $A \cup \overline{B}$.

For example the formula $a \vee b$ can be seen as rule $\leftarrow \text{not } a, \text{not } b$.

Remark 10. As seen in Example 3, we can identify a conjunction of literals as a set containing all of its literals. Therefore, we can denote $b \in B$, where B is a given body, meaning that the atom b occurs in the positive part of the body. Reversely we can write $\neg b \in B$, if the atom b occurs in the negative part of the body.

Definition 30. A (logic) *program* Π consists of finitely many rules. A program is called disjunctive when there is at least one rule in the program, that is disjunctive. Vice versa, we call a program non-disjunctive if it only consists of non-disjunctive rules. A program Π is positive, if all rules consist of the positive part of the body only.

Definition 31. By *Bodies(Π)* we will refer to all the body parts of the rules of a program Π.

Definition 32. Like in Definition 7 we denote the set of atoms occurring in the body of a rule as $atoms(B)$. By $atoms(\Pi)$ where Π is a program, we refer to all the atoms occurring in the program Π.

Example 12. An example for a program Π is

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

This program consists of four rules. The heads of these rules are $a \vee b$, $\emptyset$, c and d. The bodies are c, $\neg b$, d, $\neg e$ and e. The rule $a \vee b \leftarrow c$ is the only disjunctive rule in the program. The positive part of the body of the rule $c \leftarrow d, \neg e$ is d, the negative part of the body is $\neg e$. The atoms of the program $atoms(\Pi) = \{a, b, c, d, e\}$. The according CNF formula would be

$$\begin{aligned} a \vee b \vee \neg c \\ b \\ c \vee \neg d \vee e \\ e \vee \neg e. \end{aligned}$$

3.2 Answer Sets

3.2.1 Answer Set Programming

We recall, that answer set programming (ASP) is a form of declarative programming, where the solution of a program consisting of one or more rules is an answer set, which is a stable model. The stable model semantics of logic programming was introduced by Gelfond and Lifschitz in 1988 [GL88]. The programs that generate answer sets are called answer set solvers (cf. [Lif08]).

3.2.2 Answer Sets of a Positive Program

To get a better understanding of answer sets, we consider a positive, non-disjunctive program Π, regarding to [Lie17].

Definition 33. Let X be a set of atoms. X *satisfies a positive rule*

$$a_0 \leftarrow b_1, ..., b_l \tag{3.6}$$

of a program when $a_0 \in X$ whenever $\{b_1, ..., b_l\} \subseteq X$.

By definition, every singleton a_0 satisfies Equation 3.6. The truth table of Definition 33 matches with the truth table of the implication operator in propositional logic, where $A \leftarrow B \iff B \implies A$, see Figure 3.1. We can also view $A \leftarrow B$ as logically equivalent to $A \vee \overline{B}$.

a_1	b_1	$a_1 \leftarrow b_1$
true	true	true
false	false	true
true	false	true
false	true	false

Figure 3.1: truth table of Definition 33

Definition 34. Let X be a set of atoms. We say that X *satisfies a positive program* Π if X satisfies every rule in Π.

Proposition 4. *[Lie17] For any positive program* Π*, there exists a set of atoms satisfying that program.*

Proof. See [Lie17]. □

Proposition 5. *[Lie17] Let* Π *be a positive, non-disjunctive program. The intersection of all sets satisfying a program* Π *satisfies* Π *as well.*

Proof. See [Lie17]. □

Remark 11. Because of Proposition 5, there is a minimal set of atoms that satisfies Π.

Definition 35. The smallest set of atoms satisfying a positive, non-disjunctive program Π is called the *answer set* of Π.

Example 13. Consider the program

$$\begin{aligned} a.\\ c \leftarrow a, b. \end{aligned} \tag{3.7}$$

The sets of atoms satisfying the program in Equation 3.7 are

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

The answer set is $\{a\}$ as it is minimal amongst the sets of atoms satisfying the program.

Let Π be a program consisting of facts only. The set of these facts is indeed the only answer set of the program. A fact states what is known, while an answer set reflects this information by asserting that atoms that are a fact in the program are *true*, while everything else is $false$. Any positive program has a unique answer set. This follows from the definition of an answer set and Proposition 4.

3.2.3 Answer Sets of a Disjunctive Program

Now we look at answer sets in disjunctive programs that don't necessarily have to be positive programs.

Definition 36. Let Π be a disjunctive program consisting of rules in the form $A \leftarrow B_1, B_2$, where B_1 is the positive and B_2 denotes the negative part of the body of the rule and let X be a set of atoms. The *Gelfond-Lifschitz reduct* or just "reduct" Π^X of Π with respect to X is obtained from Π by

1. removing each rule $A \leftarrow B_1, B_2$, where there exists an atom in B_2 that is also in X, i.e. $X \cap atoms(B_2) \neq \emptyset$, and
2. replacing the body of each remaining rule $A \leftarrow B_1, B_2$ with the positive part of the body, i.e. $A \leftarrow B_1$.

Definition 37. A set of atoms X is an *answer set* of a program Π, if X is minimal among the set of atoms that satisfy the reduct Π^X. [GL88] Therefore, X is an answer set of Π if X is an answer set of the reduct Π^X. An answer set is also called a stable model.

While in the positive program in Section 3.2.2 we could say that we can generate the head a_0, as soon as we generated the atoms of the body $b_1, ..., b_l$, if none of the atoms of the negative part of the body $b_{l+1}, ..., b_m$ can be generated using the rules of the program. The answer set is the set of all atoms that can be generated by applying rules of the program in any order.

In a program with negation, the reduct ensures that we generate one of the atoms in the head as soon as we generated the positive part of the body B^+ and provides that the atoms of the negative part of the body can not be generated using the rules of the program.

Example 14. Let Π be the program

$$\begin{aligned} a &\leftarrow \neg b, \neg c \\ b &\leftarrow \neg c \\ c &\leftarrow a \\ d &\leftarrow d \\ e, f &\leftarrow \neg a, \neg b. \end{aligned} \tag{3.8}$$

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

$$\begin{aligned} b &\leftarrow \\ c &\leftarrow a \\ d &\leftarrow d. \end{aligned} \tag{3.9}$$

In the first rule of Π, we apply the first rule of the reduct and therefore the rule is being removed. In the second rule of Π, we apply the second rule of the reduct, hence the negative part of the body is being removed. The other rules remain the same.

Note that in the reduct in Equation 3.9 the first rule $b \leftarrow$ is equivalent to the fact b. We can rewrite Equation 3.9 as

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

Now we can see clearly that to satisfy the program, the atom b needs to be assigned to *true*. The reduct $\Pi^{\{b\}}$ satisfies the program in Equation 3.8 and the set $\{b\}$ is an answer set, because the only subset would be the empty set $\emptyset$, which does not satisfy the reduct, because of the fact b.

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

$$\begin{aligned} c &\leftarrow a \\ d &\leftarrow d. \end{aligned} \tag{3.10}$$

Here, the first and the second rule of Π are being removed due to rule 1 of the reduct. The reduct is being satisfied by the set $\{a, c\}$. But it is not an answer set, as it is not minimal.

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

$$\begin{aligned} a \leftarrow \\ b \leftarrow \\ c \leftarrow a \\ d \leftarrow d \\ e, f \leftarrow . \end{aligned} \tag{3.11}$$

Only rule 2 of the reduct needs to be applied to the first two rules of Π.
In this case a and b and either e or f have to be assigned to value *true*. If a is true, c needs to be *true* as well. The minimal set, satisfying the reduct is $\{a, b, c, e\}$ or $\{a, b, c, f\}$. So the empty set $\emptyset$ cannot be an answer set.

Consider the set $\{b, d\}$. The reduct $\Pi^{\{b,d\}}$ is

$$\begin{aligned} b \leftarrow \\ c \leftarrow a \\ d \leftarrow d. \end{aligned} \tag{3.12}$$

Here, the first rule is being removed due to rule 1 of the reduct. The body of the second rule of Π is removed due to rule 2 of the reduct. The reduct is being satisfied by the set $\{b, d\}$.
But the reduct $\Pi^{\{b,d\}}$ is not minimal, because the set $\{b\}$ would satisfy the reduct and is a subset of $\{b, d\}$. Therefore the set $\{b, d\}$ is not an answer set.

Example 15. If the program Π consists of the rule $a \leftarrow \neg a$ then there is no answer set at all.

If we look at the possibilities, we know, that we either can assign the atom a to the value *true* or to the value $false$.

In the first case, let a be *true*. The reduct $\Pi^{\{a\}}$ is a program consisting of no rule. The minimal set that would satisfy the empty program is the empty set. Thus, the set $\{a\}$ is not an answer set.

In the second case, let a be $false$. There are no atoms that were assigned value true, therefore the equivalent reduct is $\Pi^{\{\emptyset\}}$. The reduct $\Pi^{\{\emptyset\}}$ is

$$a \leftarrow,$$

which is equivalent to

$$a.$$

The empty set $\emptyset$ does not satisfy the reduct and is not the answer set of the program.

3.2.4 Classification of models

Up to this point, when talking about a model, we were talking about classical models that satisfy formulas and in ASP we compute stable models. As we will take a further step towards abstract ASP, we will give an overview of three types of models: classical models, supported models and stable models.

Definition 38. A *supporting rule* for an atom a with respect to a set of literals L is a rule of a program Π of the form

$$A \vee a \leftarrow B,$$

where holds

$$L \cap (\overline{B} \cup A) = \emptyset.$$

We say, that a rule is a supporting rule for an atom a with respect to a set of literals L if a is in the head of a rule and there holds that the following sets are disjunct:

- the set of literals L and
- the clause we identify the rule with, if we removed the atom a from the head. (by using Remark 3.5)

Example 16. Consider the program of Example 15. The rule $a \leftarrow \neg a$ for an atom a with respect to the empty set of literals $\emptyset$ is a supporting rule, because it holds $\emptyset \cap \{a \cup a\} = \emptyset$. The rule $a \leftarrow \neg a$ is not a supporting rule for the atom a with respect to a set of literals a, because there does not hold

$$L \cap (\overline{B} \cup A) = \{a\} \cap (\{a\} \cup \{a\}) = \{a\} \cap \{a\} = \{a\} \neq \emptyset.$$

Definition 39. Let L be a consistent and complete set of literals over $atoms(\Pi)$ of a program Π. We call L

- a *classical* model of Π, if L satisfies every rule in Π.
- a *supported* model of Π if L is a classical model of Π and for every atom that occurs positively in L, i.e. $a \in L^+$ there is a supporting rule for a with respect to L.
- a *stable* model of a program Π if the set L^+ is an answer set of Π.

Definition 40. The *completion* $comp(\Pi)$ of a program Π consists of Π and the formulas

$$\Big\{\neg a \vee \bigvee_{A \vee a \leftarrow B \in \Pi} (B \wedge \overline{A} | a \in atoms(\Pi)\Big\}.$$

Example 17. Assume a program consisting of the rule $a \vee b \leftarrow c$.
The set of literals $\{a, b, c\}$ satisfies the rule, because $a \vee b \vee \neg c$ holds. But it does not satisfy the completion, as $\{\neg a \vee (c \wedge \neg b)\}$ does not hold.

Definition 41. *[LRS97]* Let Π be a program and L be a consistent set of literals over $atoms(\Pi)$. A set of atoms over $atoms(\Pi)$ X is *unfounded* on L with respect to Π, when for each atom $a \in X$ and each rule $A \leftarrow B \in \Pi$ such that $a \in A$, either one of the following conditions hold

- $L \cap \overline{B} \neq \emptyset$,
- $X \cap B \neq \emptyset$ or
- $(A\ X) \cap L \neq \emptyset$.

An important theorem that helps understanding key computational ideas behind modern ASP solvers stems from [LRS97]:

Proposition 6. *[BLM16] For a program* Π *and a consistent and complete set* L *of literals over atoms*(Π), L *is a stable model of* Π *if and only if* L *is a classical model of* Π *and no non-empty subset of* L^+ *is an unfounded set on* L *with respect to* Π.

4 Abstract ASP with Backtracking

Like in the case of DPLL, we are making an abstract approach to ASP. We do this for the same reasons - better analysability, comparability between solvers, provability of the correctness as well as it is easier to design new algorithms. We recall that only five answer set systems are capable of solving disjunctive programs: DLV, GNT, CMODELS, CLASP, WASP. In [BLM16], the authors built a framework to capture multiple solvers at once. We will take a closer look at DLV, GNT and CMODELS to understand the rules and build the base for extending the work of the authors with the previously introduced transition rules of backjumping and learning. Further, we will need this to understand the general framework capturing all these solvers, that was being introduced in [BLM16], which we will also be extending in the next chapter. Most of the background information in Chapter 4.2 is taken from [Lie11], while the rest is based on [BLM16].

4.1 A Two-Layer Abstract Solver

A popular approach for disjunctive answer set solvers is the use of two layers: The `generate layer` is used to find potential answer sets of a given program. By using transition rules, a set of candidates is being obtained. The `test layer` determines whether the outcome of the generate layer is indeed an answer set of the program.

The idea is to design abstract solvers made of two graphs, which are the layers. We can construct a graph $DP^2_{g,t}(\Pi)$ that captures these two-layer technology, where the two layers of the graphs communicate with each other via transition rules.

4.2 Abstract CMODELS

CMODELS is one of the systems that computes answer sets, where the algorithm does not explore the same part of the search tree more than once. The algorithm is called ASP-SAT with Learning, according to [GLM06]. In this solver, a SAT-based approach is being followed, as a SAT solver is being used as search engine. The two-layer architecture is based on the DPLL procedure for both, the generating and the test layer, where the generate layer is the completion of the program Π converted to conjunctive normal form and the test layer is the conjunction of all so called "loop formulas" of Π. They showed that a set is an answer set if and only if it satisfies its completion and the set of all loop formulas, as they found out that cycles are the reason why models of a logic program's completion may not be answer sets, but we won't go into depth there. For further information see [LZ02].

CMODELS tried to overcome the disadvantages of Lin and Zhao's answer set solver ASSAT, which can only compute one answer set and works only with basic rules. Transition rules

like *UnitDPL*, *Success*, *Backjump*, *Decide* and *Conclude* of graph DPL_F are referred to as basic, while *Learn* is not.

Like in DPLL, we want to take an abstract approach to CMODELS. In a first step, we look at a graph $\mathrm{DP}^2_{g,t}(\Pi)$ of the two-layer architecture, where we describe a two-layer solver using DPLL, which is exactly what the solver CMODELS is based on. We will look at the generate and test layer of CMODELS in particular.

Later, in Chapter 5.1, we will take the graph that we reviewed here and extend it to backjumping and learning.

4.2.1 An abstract two-layer solver via DPLL

To create a graph $\mathrm{DP}^2_{g,t}(\Pi)$ with two layers that captures CMODELS, we need to determine the nodes and the edges accordingly by extending the definition of a state to capture generate and test layers. For this, we need to define a label first.

Definition 42. A `label` is either the symbol $\mathcal{L}$ or $\mathcal{R}$.

Definition 43. [BLM16] A *state* relative to sets X and X' of atoms is either

1. a pair $(L, R)_s$, where L and R are records relative to X and X', respectively, and s is a label ($s \in \{\mathcal{L}, \mathcal{R}\}$).
2. $Ok(L)$, where L is a record relative to X, or
3. the distinguished state $Failstate$.

Definition 44. A function g from a program to another program is called a *generating (program) function*, if there holds $atoms(\Pi) \subset atoms(g(\Pi))$ for any program Π.

Definition 45. Let Π be a program and Π' be a non-disjunctive program. Let M be a consistent set of literals covering the program Π. A function $t : (\Pi, M) \to \Pi'$ is called a *witness (program) function* with respect to Π and M. We denote

$$\mathrm{atoms}(t, \Pi, X) = \bigcup_L \mathrm{atoms}(t(\Pi, L)), \tag{4.1}$$

for all possible consistent and complete sets L of literals over X, where Π is a given program and t is a given witness function.

Now we have everything to construct the graph $\mathrm{DP}^2_{g,t}(\Pi)$.
The nodes of the graph $\mathrm{DP}^2_{g,t}(\Pi)$, where g is a generating function, t is a witness function and Π is a program, are the states relative to the sets $atoms(g(\Pi))$ and $atoms(t, \Pi, atoms(g(\Pi))))$, respectively. The initial state is $(\emptyset, \emptyset)_{\mathcal{L}}$.

The edges of the graph are determined by the transition rules in the Figures 4.1, 4.2, 4.3. The transition rules are being partitioned in Left-rules, Right-rules and Crossover-rules. The rules are based on the classical DPLL transition rules. Especially, the left-rule $Conclude_{\mathcal{L}}$, both backtracking rules $Backtrack_{\mathcal{L}}$ and $Backtrack_{\mathcal{R}}$ remain in nearly the same syntax as in DP_F.

Left-rules

$$Conclude_{\mathcal{L}} \quad (L,\emptyset)_{\mathcal{L}} \quad \Longrightarrow Failstate \text{ if } \begin{cases} L \text{ is inconsistent and} \\ L \text{ contains no decision literal} \end{cases}$$

$$Backtrack_{\mathcal{L}} \quad (Ll^{\Delta}L',\emptyset)_{\mathcal{L}} \Longrightarrow (L\bar{l},\emptyset)_{\mathcal{L}} \quad \text{if } \begin{cases} Ll^{\Delta}L' \text{ is inconsistent and} \\ L' \text{ contains no decision literal} \end{cases}$$

$$Unit_{\mathcal{L}} \quad (L,\emptyset)_{\mathcal{L}} \quad \Longrightarrow (Ll,\emptyset)_{\mathcal{L}} \quad \text{if } \begin{cases} l \text{ is a literal over } atoms(g(\Pi)) \text{ and} \\ l \text{ does not occur in } L \text{ and} \\ \text{a rule in } g(\Pi) \text{ is equivalent to } C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$$

$$Decide_{\mathcal{L}} \quad (L,\emptyset)_{\mathcal{L}} \quad \Longrightarrow (Ll^{\Delta},\emptyset)_{\mathcal{L}} \quad \text{if } \begin{cases} L \text{ is consistent and} \\ l \text{ is a literal over } atoms(g(\Pi)) \text{ and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } L \end{cases}$$

Figure 4.1: The left-rules of the graph $DP^2_{g,t}(\Pi)$.

Right-rules

$$Conclude_{\mathcal{R}} \quad (L,R)_{\mathcal{R}} \quad \Longrightarrow Ok(L) \quad \text{if } \begin{cases} R \text{ is inconsistent and} \\ R \text{ contains no decision literal} \end{cases}$$

$$Backtrack_{\mathcal{R}} \quad (L,Rl^{\Delta}R')_{\mathcal{R}} \Longrightarrow (L,R\bar{l})_{\mathcal{R}} \quad \text{if } \begin{cases} Rl^{\Delta}R' \text{ is inconsistent and} \\ R' \text{ contains no decision literal} \end{cases}$$

$$Unit_{\mathcal{R}} \quad (L,R)_{\mathcal{R}} \quad \Longrightarrow (L,Rl)_{\mathcal{R}} \quad \text{if } \begin{cases} l \text{ is a literal over } atoms(t(\Pi,L)) \text{ and} \\ l \text{ does not occur in } R \text{ and} \\ \text{a rule in } t(\Pi,L) \text{ is equivalent to } C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$$

$$Decide_{\mathcal{R}} \quad (L,R)_{\mathcal{R}} \quad \Longrightarrow (L,Rl^{\Delta})_{\mathcal{R}} \quad \text{if } \begin{cases} R \text{ is consistent and} \\ l \text{ is a literal over } atoms(t(\Pi,L)) \text{ and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } R \end{cases}$$

Figure 4.2: The right-rules of the graph $DP^2_{g,t}(\Pi)$.

Crossing-rule $\mathcal{LR}$

$Cross_{\mathcal{LR}}$ $\quad (L, \emptyset)_{\mathcal{L}} \quad \Longrightarrow (L, \emptyset)_{\mathcal{R}}$ $\quad$ if { no left-rule applies

Crossing-rules $\mathcal{RL}$

$Conclude_{\mathcal{RL}}$ $\quad (L, R)_{\mathcal{R}} \quad \Longrightarrow Failstate$ if $\begin{cases} \text{no right-rule applies and} \\ L \text{ contains no decision literal} \end{cases}$

$Backtrack_{\mathcal{RL}}$ $(Ll^{\Delta}L', R)_{\mathcal{R}} \Longrightarrow (L\bar{l}, \emptyset)_{\mathcal{L}}$ if $\begin{cases} \text{no right-rule applies and} \\ L' \text{ contains no decision literal} \end{cases}$

Figure 4.3: The crossing rules of the graph $DP^2_{g,t}(\Pi)$.

The left-rules of $\mathrm{DP}^2_{g,t}(\Pi)$ capture the generate layer of the two layers. The program Π is transformed to a program $g(\Pi)$ by using the generating function. Then, the DPLL procedure is applied to the new program $g(\Pi)$.

The right-rules of $\mathrm{DP}^2_{g,t}(\Pi)$ capture the test layer. Here, we consider the witness program and we again apply the right-rules to the transformed program $t(\Pi, M)$.

The label $\mathcal{L}$ indicates that the current computation is within the generate layer, the label $\mathcal{R}$ indicates that the current computation is within the test layer. These labels provide us with information about the previous rule, that we applied, and what rules we are allowed to apply next. A state with the label $\mathcal{L}$ (respectively $\mathcal{R}$) suggests that the last rule we applied to this state was one of the left-rules (right-rules) or a crossing-rule $\mathcal{RL}$ ($\mathcal{LR}$). Therefore, we can apply one of the left-rules (right-rules) or the crossing-rule $\mathcal{LR}$ ($\mathcal{RL}$) in the next step. The left-hand side L of the state $(L, R)_s$ belongs to the generate layer and records the state of computation. The right-hand side R of the state $(L, R)_s$ records the computation state due to the test layer, respectively.

Note that now the left-rule $Conclude_{\mathcal{R}}$ does not lead to $Failstate$ as before, but to $Ok(L)$.

As we consider an extended program consisting of the atoms $atoms(g(\Pi))$, that was being received using the generating function, we need to slightly adapt the left-rules $Unit_{\mathcal{L}}$ and $Decide_{\mathcal{L}}$ compared to the rules of DP_F and its variations,, so we check the added rules as well. Other than that the rules are similar to the classical case.

Analogously, using a right-rule, we consider the extended program consisting of the atoms $atoms((t(\Pi, L)))$, that was being received using the witness function. Here we need to adapt the right-rules $Unit_{\mathcal{R}}$ and $Decide_{\mathcal{R}}$ as well.

The crossing-rules give us a method to change between left- and right-rules. In the case of crossing from $\mathcal{L}$ to $\mathcal{R}$ we just switch the state whenever we find a model of $g(\Pi)$ (see Theorem 1). In the case of crossing from $\mathcal{R}$ to $\mathcal{L}$ we will either find out that there is no answer set in our program and we will stop the procedure or we need to change the model on the left side L and find a new model of our program, that is extended through the generating function, that could lead to an unsatisfiable right side R.

To sum things up, we start the procedure at the initial state$(\emptyset, \emptyset)_{\mathcal{L}}$, which suggests that we find ourselves in the generate layer. Left-rules are being applied until we either find a (classical) model of the generating program $g(\Pi)$ or we arrive at $Failstate$. If we found a model, there are no more left-rules that can be applied. Therefore, we apply $Cross_{\mathcal{LR}}$. Then, we consider a witness program with respect to L, $t(\Pi, L)$. When there is no classical model for the witness program, then $Conclude_{\mathcal{R}}$ can be applied, which leads to a terminal state $Ok(L)$. This would mean that L is a solution to a given search problem. If there is no right-rule that can be applied on a state $(L, R)_{\mathcal{R}}$, R is a (classical) model of the witness program. Then, we have to apply the one of the crossing-rules $\mathcal{RL}$. Either there is a chance to find a new model of $g(\Pi)$ by backtracking through $Backtrack_{\mathcal{RL}}$ or we will arrive at $Failstate$.

We describe some special features of the graph in the following theorem:

Theorem 1. *[BLM16, Proposition 2] For a disjunctive program Π, a generating function g and a witness function t, there holds*

1. *the graph $DP^2_{g,t}$ is finite and acyclic,*
2. *any terminal state of $DP^2_{g,t}$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L being a classical model of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable,*
3. *$Failstate$ is reachable from the initial state if and only if $g(\Pi)$ has no model such that its witness is unsatisfiable.*

Proof. See [BLM16, Proposition 2]. □

Direct Proof of Finiteness. To show, that the graph $\mathrm{DP}^2_{g,t}(\Pi)$ is finite, we consider some state $(L, R)_s$ of $\mathrm{DP}^2_{g,t}(\Pi)$.

Assume $s = L$. L is a record to $atoms(g(\Pi))$, where $g(\Pi)$ is the generating function applied to the program. The program Π is bounded by its finitely many rules with finitely many atoms in the head and in the body. Thus, L is bounded by the size of the program Π. Also, L does not allow repetitions, as it is a record. By applying the generating function, we remain finite, as the function g maps one program to another program. Thus, there is a finite number of possible strings L.

Using the same argument, we see that $s = R$ is a record to $atoms(t, \Pi, atoms(g(\Pi)))$. From the first argument we know that $atoms(g(\Pi))$ is finite, $atoms(\Pi)$ is finite per definition and t is a function from a program to another program and therefore finite. Combining them will again lead to a finite set of atoms.

There is a finite amount of labels for the state $\mathrm{DP}^2_{g,t}(\Pi)$, which is either $\mathcal{L}$ or $\mathcal{R}$. All possible outcomes of L and R are finite and we have a finite amount of labels. Putting it all together, we receive a finite amount of states. Therefore, the graph $\mathrm{DP}^2_{g,t}(\Pi)$ is finite. □

4.2.2 The two layers of CMODELS

Now that we have constructed the abstract two-layer solver via DPLL, we take a closer look at CMODELS by reviewing the generating function g^C and the witness function t^C to understand how such a solver can look in practice. The according theory is being taken from [BLM16]. The following definition does not directly consider loop formulas, but deals with minimal models.

For the generate layer, CMODELS is using $g^C(\Pi)$, which corresponds to the completion of the program Π. Then, the DPLL procedure is being applied to $g^C(\Pi)$.

For the test layer, CMODELS is depending on the program produced by the witness function t^C, that tests the minimality of the found models of the completion.

For the construction of g^C, we use an auxiliary atom α_B for every body B in the program Π. For B there holds the following:

$$\alpha_B = \begin{cases} true, & \text{if } B \text{ is true} \\ false, & \text{else} \end{cases}$$

In addition, an auxiliary atom $\alpha_{a,B}$ is created for every atom a in the head A of a disjunctive rule of the form $A \leftarrow B$. There holds

$$\alpha_{a,B} = B \wedge \overline{(A \backslash \{a\})}^{\vee}.$$

The definition of $g^C(\Pi)$ and $t^C(\Pi)$ for a program Π can be found in the Equations 4.2 and 4.3. In the definition of $g^C(\Pi)$, one can observe, that the introduced definitions α_B and $\alpha_{a,B}$ affect the first four lines of the definition, while the other two lines encode the completion with use of α_B and $\alpha_{a,B}$.

$$\begin{aligned} g^C(\Pi) = \quad & \{\alpha_B \vee \overline{B} | B \in Bodies(\Pi)\} \\ & \{\neg\alpha_B \vee a | B \in Bodies(\Pi), a \in B\} \\ & \{\alpha_{a,B} \vee \neg\alpha_B \vee A | A \vee a \leftarrow B \in \Pi\} \\ & \{\neg\alpha_{a,B} \vee b | A \vee a \leftarrow B \in \Pi, b \in \overline{A} \cup \{\alpha_B\}\} \\ & \{\neg\alpha_B \vee A | A \leftarrow B \in \Pi\} \\ & \{\neg\alpha \bigvee_{a \leftarrow B \in \Pi} \alpha_B \bigvee_{A \vee a \leftarrow B \in \Pi} \alpha_{a,B}\} \end{aligned} \tag{4.2}$$

Intuitively, CMODELS uses $g^C(\Pi)$ to approximate the program Π. Any stable model of Π is a classical model of $g^C(\Pi)$. The reverse does not have to be true. Therefore, we need to check whether a classical model of $g^C(\Pi)$ is a stable models of Π by using the test layer.

$$\begin{aligned} t^C(\Pi, M) = \quad & \{\overline{M^+_{|atoms(\Pi)}}^{\vee}\} \cup \\ & \{\neg a | \neg a \in M_{|atoms(\Pi)}\} \cup \\ & \{\overline{B} \vee A | A \leftarrow B \in \Pi^{M^+}, B \subseteq M\} \end{aligned} \tag{4.3}$$

Let M be a classical model of $g^C(\Pi)$. M is a stable model of Π if and only if a program produced by the witness function $t^C(\Pi, M)$ has no classical models. Any model N of $t^C(\Pi, M)$ satisfies the reduct Π^{M^+}, while $N^+ \subseteq M^+_{|atoms(\Pi)}$. Then $M^+_{|atoms(\Pi)}$ is not an answer set of Π, as it is not minimal, and therefore M is not a stable model of Π.

4.3 Abstract GNT

The disjunctive solver GNT (Generate'n'Test) is an experimental implementation of the stable model semantics for disjunctive logic programs, see [Jan]. It was being introduced in 2006 by Janhunen et al. [JNS+06]. In Chapter 4.2, we have seen that the solver CMODELS uses the DPLL procedure for generating and testing. In GNT, two SMODELS solvers are involved instead. The purpose of these solvers is similar to CMODELS: One SMODELS solver instance is being used for generating model candidates, the other for checking minimality.

For a better understanding, we will take a look at SMODELS first. Then, we will use it to give an overview of GNT. Similarly to CMODELS, we will take a look at the graph and its layers. In Chapter 5.2 we will extend the graph to backjumping and learning.

4.3.1 Abstract Solver via SMODELS

SMODELS is an algorithm for finding answer sets of non-disjunctive logic programs. To understand abstract GNT we need to take a look at abstract SMODELS . In [Lie11], the author introduces the graph SM_Π, where the terminal nodes are answer sets of the program Π. The nodes are the states relative to the sets of atoms occurring in Π. The transition rules of the new graph are partly being taken from the graph DP_F, namely *Decide*, *Conclude*, *Backtrack* and *Success*. The other transition rules for the non-disjunctive case are presented in Figure 4.4 according to [BLM16]. The SMODELS procedure finds stable models for non-disjunctive programs, while the DPLL procedure finds classical models.

UnitPropagate :

$$L \Longrightarrow Ll \qquad \text{if} \begin{cases} \text{a rule in } \Pi \text{ that is equivalent to } C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$$

AllRulesCancelled :

$$L \Longrightarrow L\bar{l} \qquad \text{if } \{ \text{there is no rule in } \Pi \text{ supporting } l \text{ with respect to } L$$

BackchainTrue :

$$L \Longrightarrow Ll \qquad \text{if} \begin{cases} \text{there is a rule } A \vee a \leftarrow B \text{ in } \Pi \\ \text{so that (i) } a \in L, \text{ and (ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and,} \\ \text{(iii) no other rule in } \Pi \text{ is supporting } a \text{ with respect to } L \end{cases}$$

Unfounded :

$$L \Longrightarrow L\bar{l} \qquad \text{if} \begin{cases} L \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } l \text{ such that} \\ X \text{ is unfounded on } L \text{ with respect to } \Pi \end{cases}$$

Figure 4.4: Transition rules of the graph SM_Π.

Remark 12. Note that the rule *UnitPropagate* is syntactically the same as the rule *Unit*.

We only transformed the notation from clauses to rules.

In [Lie11], there is a proof for some features for the non-disjunctive case.

Proposition 7. *[Lie11] For a non-disjunctive program* Π*,*

1. *graph* SM_Π *is finite and acyclic,*
2. *for any terminal state* M *of* SM_Π *other than* $Failstate$*,* M^+ *is an answer set of* Π*,*
3. $Failstate$ *is reachable from* $\emptyset$ *in* SM_Π *if and only if* Π *has no answer sets.*

Proof. see [Lie11]. □

As SMODELS is not capable of capturing disjunctive programs, we will now proceed with GNT, which is based on SMODELS and can capture disjunctive programs.

4.3.2 The graph $SM^2_{g(\Pi),t}$

As GNT, similarily to CMODELS, uses the two-layer approach as well, we will review the new rules on a two-layer basis according to [BLM14].

We will construct a graph $SM^2_{g(\Pi),t}$ to capture GNT. The nodes of the graph $SM^2_{g(\Pi),t}$ are the states and the edges consist of the transition rules of SMODELS, where we introduce the generate and test layer like in $DP^2_{g,t}(\Pi)$ in Section 4.2.1 and add the crossing rules.

The left-rules of $SM^2_{g(\Pi),t}$ are then $Decide_\mathcal{L}$, $Backtrack_\mathcal{L}$, $Unit_\mathcal{L}$ and $Conclude_\mathcal{L}$ from Figure 4.1, as well as $AllRulesCancelled_\mathcal{L}$, $BackchainTrue_\mathcal{L}$ and $Unfounded_\mathcal{L}$ from Figure 4.5.

The right-rules are $Decide_\mathcal{R}$, $Backtrack_\mathcal{R}$, $Unit_\mathcal{R}$ and $Conclude_\mathcal{R}$ from Figure 4.2 and $AllRulesCancelled_\mathcal{R}$, $BackchainTrue_\mathcal{R}$ and $Unfounded_\mathcal{R}$ from Figure 4.6.

The crossing-rules consist of $Conclude_{\mathcal{RL}}$, $Cross_{\mathcal{LR}}$ and $Backjump_{\mathcal{RL}}$ from Figure 4.3.

Left-rules

$AllRulesCancelled_\mathcal{L}$:

$$(L,\emptyset)_\mathcal{L} \Longrightarrow (L\bar{l},\emptyset)_\mathcal{L} \text{ if } \{ \text{ there is no rule in } g(\Pi) \text{ supporting } l \text{ with respect to } L$$

$BackchainTrue_\mathcal{L}$:

$$(L,\emptyset)_\mathcal{L} \Longrightarrow (Ll,\emptyset)_\mathcal{L} \text{ if } \begin{cases} \text{there is a rule } A \vee a \leftarrow B \text{ in } g(\Pi) \\ \text{so that (i) } a \in L, \text{ and (ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and,} \\ \text{(iii) no other rule in } g(\Pi) \text{ is supporting a with respect to } L \end{cases}$$

$Unfounded_\mathcal{L}$:

$$(L,\emptyset)_\mathcal{L} \Longrightarrow (L\bar{l},\emptyset)_\mathcal{L} \text{ if } \begin{cases} L \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } l \text{ such that} \\ X \text{ is unfounded on } L \text{ with respect to } g(\Pi) \end{cases}$$

Figure 4.5: Left rules of the graph $SM^2_{g(\Pi),t}$.

Right-rules

$AllRulesCancelled_{\mathcal{R}}$:

$$(L,R)_{\mathcal{R}} \Longrightarrow (L,R\bar{l})_{\mathcal{R}} \text{ if } \{\text{there is no rule in } t(\Pi,L) \text{ supporting } l \text{ with respect to } R$$

$BackchainTrue_{\mathcal{R}}$:

$$(L,R)_{\mathcal{R}} \Longrightarrow (L,Rl)_{\mathcal{R}} \text{ if } \begin{cases} \text{there is a rule } A \vee a \leftarrow B \text{ in } t(\Pi,L) \\ \text{so that (i) } a \in R, \text{ and (ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and,} \\ \text{(iii) no other rule in } t(\Pi,L) \text{ is supporting a with respect to } R \end{cases}$$

$Unfounded_{\mathcal{R}}$:

$$(L,R)_{\mathcal{R}} \Longrightarrow (L,R\bar{l})_{\mathcal{R}} \text{ if } \begin{cases} R \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } l \text{ such that} \\ X \text{ is unfounded on } R \text{ with respect to } t(\Pi,L) \end{cases}$$

Figure 4.6: Right rules of the graph $SM^2_{g(\Pi),t}$.

4.3.3 The two layers of GNT

In [JNS+06], the generating function g^G and the testing function t^G were defined by the authors. For the definitions, we need to divide the disjunctive program Π into a part consisting of all the non-disjunctive rules of Π, which is being called Π_N. The set of disjunctive rules $\Pi \backslash \Pi_N$ is bein called Π_D. Additionally, for each atom a in $atoms(\Pi)$ two new atoms a^r and a^s are being introduced.

The generating function is defined as

$$\begin{aligned} g^G(\Pi) = \ & \{a \leftarrow B, \neg a^r | A \vee a \leftarrow B \in \Pi_D\} \cup \\ & \{a^r \leftarrow \neg a \mid A \vee a \leftarrow B \in \Pi_D\} \cup \\ & \{\leftarrow \overline{A}, B \mid A \leftarrow B \in \Pi_D\} \cup \\ & \Pi_N \cup \\ & \{a^s \leftarrow \overline{A \backslash \{a\}}, B \mid A \vee a \leftarrow B \in \Pi_D\} \cup \\ & \{\leftarrow a, \neg a^s \mid a \vee A \leftarrow B \in \Pi_D\}, \end{aligned} \tag{4.4}$$

The definition of the testing function can be found in Equation 4.5.

$$\begin{aligned} t^G(\Pi, M) = \ & \{a \leftarrow B, \neg a^r \mid A \vee a \leftarrow B \in \Pi_D^M, a \in M, B \subseteq M\} \cup \\ & \{a^r \leftarrow \neg a \mid A \vee a \leftarrow B \in \Pi\} \cup \\ & \{\leftarrow \overline{A}, B \mid A \leftarrow B \in \Pi_D^M, B \subseteq M\} \cup \\ & \{a \leftarrow B \mid a \leftarrow B \in \Pi_N^M, a \in M, B \subseteq M\} \cup \\ & \{\leftarrow M_{|atoms(\Pi)}\} \end{aligned} \tag{4.5}$$

4.4 Abstract solver DLV

A third answer set solver, that is some form of blend between the previous solvers GNT and CMODELS, is called DLV, which is being introduced in [LFP+06]. The generate layer resembles the SMODELS algorithm, but the rule *Unfounded* is not used. The test layer is using the DPLL procedure. Knowing this, it makes sense to capture this solver using the graph $(SM^* \times DP)_{g(\Pi),t}$, where the asterisk symbol emphasizes that the generate layer is not quite the same as in SMODELS, but still similar to it. Another difference to SMODELS will be made clear when looking at the generate layer in Section 4.4.2 as SMODELS cannot deal with disjunctive programs, while DLV can.

In [BLM14] and [BLM16], the authors introduced abstract DLV without backjumping. Again, we will review the work they did in this section and extend this to backjumping and learning in Chapter 5.3.

4.4.1 The graph $(SM^* \times DP)_{g(\Pi),t}$

As the graph $(SM^* \times DP)_{g(\Pi),t}$ already describes how the layers are structured, we quickly see that the generate layer, which is represented by $SM*$, consists of the same left-rules as $SM^2_{g(\Pi),t}$ without the transition rule $Unfounded_{\mathcal{L}}$.

DLV handles disjunctive rules directly, which we will see when inspecting the instantiation of the generate layer in Section 4.4.2. Therefore, it is important to make sure that the conditions of the transition rules are stated in a way that can deal with disjunctive rules. Looking at the generate layer in Figure 4.5 and Figure 4.2 makes clear that the conditions are not restricted to non-disjunctive rules, so we do not need to change the rules explicitely.

The test layer of $(SM^* \times DP)_{g(\Pi),t}$ is represented by the expression DP, meaning that this layer is being depicted by the DPLL procedure. The right-rules of $(SM^* \times DP)_{g(\Pi),t}$ are the right-rules of $DP^2_{g,t}(\Pi)$, which can be found in Figure 4.1.

The crossing-rules remain the same as in CMODELS and GNT, see Figure 4.3.

Both, the right-rules as well as the crossing-rules are stated in a way that can handle disjunctive programs, so we do not need to change them.

Alternatively, we can say that the transition rules are basically the rules of $DP^2_{g,t}(\Pi)$ in Figure 4.1, 4.2 and 4.3 plus the rules from Figure 4.5 without the rule $Unfounded_{\mathcal{L}}$.

The nodes of the graph $(SM^* \times DP)_{g(\Pi),t}$ are the states that can be received using the set $atoms(\Pi)$ for the left side of the layer and the union of $atoms(g(L))$ for the right side, where g is the generating function and L is the left part of the state $(L, R)_s$.

4.4.2 The two layers of DLV

Like in CMODELS and in GNT, we can have a closer look at the generating function g^D, as well as the witness function t^D to see how they are being used in practice. Again, the

theory is taken from [BLM16]. The generating function

$$g^D(\Pi) = \Pi \tag{4.6}$$

is the identity function and can therefore be omitted. DLV is special in a way that it can deal directly with a disjunctive program and does not convert it to a non-disjunctive program first. The more important it is that the conditions of the transition rules are stated in a way that can deal with disjunctive rules.
The witness function is defined as

$$\begin{aligned} t^D(\Pi, M) = \quad & \{(B \cap M^+)^\vee \vee \overline{A'}^\vee \mid A \leftarrow B \in \Pi^{M^+}, B \subseteq M, A' = A \cap M^+\} \cup \\ & \{(M_{|atoms(\Pi)})^\vee\}. \end{aligned} \tag{4.7}$$

5 Abstract ASP with Backjumping and Learning

Now, as we have reviewed the structure of abstract CMODELS, GNT and DLV with backtracking, we can extend these to backjumping and learning.

In Section 2.2.2, we have seen how the rules work for one layer. Now, we try to adapt this mechanism to the two-layer abstract solvers that we have seen by adding a *Backjump* and a *Learn* rule to the graphs.

We also want to look at some of the features that we will obtain by extending the graph. We will give a proof of those features for CMODELS and state some features for GNT and DLV.

In the next chapter, we will describe the features again for the framework that includes all three solvers. There, we will prove these features for the remaining solvers at once.
In this chapter, we will also talk about adding some additional rules to the graph, that we have seen in Section 2.2.3, namely *Forget* and *Restart*.

To extend the graph, we will define the extended state to describe the form that we will be using.

Definition 46. An *extended state* relative to sets X and X' of atoms is either

1. a pair $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$, where $(L, R)_s$ is the state described in Definition 43, $\Gamma_{\mathcal{L}}$ is a set of clauses over atoms of $g(\Pi)$ that is entailed by $g(\Pi)$ and $\Gamma_{\mathcal{R}}$ is a set of clauses over atoms of $t(\Pi, L)$ that is entailed by $t(\Pi, L)$,
2. $Ok(L)$, where L is a record relative to X, or
3. the distinguished state $Failstate$.

To make the following chapter more readable, we will write state instead of extended state. Whenever we talk about a graph with backjumping and learning a state will refer to an extended state.

5.1 Extending CMODELS to Backjumping and Learning

For CMODELS, we reviewed the graph $\mathrm{DP}^2_{g,t}(\Pi)$ in the previous chapter 4.2.

Now, we will do the same as we did with graph DPL_F, where we extended, added or removed some transition rules taken from DP_F, to create a graph $\mathrm{DPL}^2_{g,t}(\Pi)$ with backjumping and learning out of $\mathrm{DP}^2_{g,t}(\Pi)$.

Again, $g(\Pi)$ and $t(\Pi, L)$ will refer to the programs we receive by using the generating and the witness function that are given, respectively.

We will also use the extended state that we defined in Definition 46, so we are able to consider learned information, when applying one of the rules.
We find the new transition rules in Figure 5.1.

Looking at these transition rules, we see how the transition rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ relate to the graph $\mathrm{DP}^2_{g,t}(\Pi)$:

Let s and t be labels, where $s \neq t$. The transition rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ relate to the transition rules of the graph $DP^2_{g,t}(\Pi)$ in one the following ways:

$$\Big((L,R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L',R')_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})\Big) \Leftrightarrow \Big((L,R)_s \Longrightarrow (L',R')_s\Big) \tag{5.1}$$

$$\Big((L,R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow Failstate\Big) \Leftrightarrow \Big((L,R)_s \Longrightarrow Failstate\Big) \tag{5.2}$$

$$\Big((L,R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow Ok(L)\Big) \Leftrightarrow \Big((L,R)_s \Longrightarrow Ok(L)\Big) \tag{5.3}$$

$$\Big((L,R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L',R')_t \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})\Big) \Leftrightarrow \Big((L,R)_s \Longrightarrow (L',R')_t\Big) \tag{5.4}$$

The left-rules of $\mathrm{DPL}^2_{g,t}(\Pi)$ consist of the extended version of $Conclude_{\mathcal{L}}$ according to Equation 5.2 and $Decide_{\mathcal{L}}$ according to Equation 5.1, and the new rules $Backjump_{\mathcal{L}}$, $Learn_{\mathcal{L}}$ and $UnitDPL_{\mathcal{L}}$ in the form of Figure 5.1.

The right-rules of $\mathrm{DPL}^2_{g,t}(\Pi)$ consist of an extended version of $Conclude_{\mathcal{R}}$ according to Equation 5.3 and $Decide_{\mathcal{R}}$ according to Equation 5.1, and the new rules $Backjump_{\mathcal{R}}$, $Learn_{\mathcal{R}}$ and $UnitDPL_{\mathcal{R}}$ in the form of Figure 5.1.

The crossing-rules $\mathcal{LR}$ from the generate layer to the test layer becomes an extended version of the rule $Cross_{\mathcal{LR}}$ from $\mathrm{DP}^2_{g,t}(\Pi)$ according to Equation 5.4 and the crossing-rules $\mathcal{RL}$ in the other direction consist of $Conclude_{\mathcal{RL}}$ of $\mathrm{DP}^2_{g,t}(\Pi)$ according to Equation 5.2 and the new rule $Backjump_{\mathcal{RL}}$ in the form of Figure 5.4.

The new initial state is $(\emptyset,\emptyset)_{\mathcal{L}} \| (\emptyset,\emptyset)$.

Left Rules

$Backjump_{\mathcal{L}}$:

$$(Ll^{\Delta}L', \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \Longrightarrow (L\bar{l}, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \quad \text{if} \begin{cases} Ll^{\Delta}L' \text{ is inconsistent and} \\ g(\Pi) \models \bar{l} \vee \overline{L} \end{cases}$$

$Learn_{\mathcal{L}}$:

$$(L, \emptyset) \| (\Gamma_{\mathcal{L}}, \emptyset) \Longrightarrow (L, \emptyset) \| (C \cup \Gamma_{\mathcal{L}}, \emptyset) \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in } g(\Pi) \text{ and} \\ g(\Pi) \models C \end{cases}$$

$UnitDPL_{\mathcal{L}}$:

$$(L, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \Longrightarrow (Ll, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \quad \text{if} \begin{cases} l \text{ is a literal over } atoms(g(\Pi))) \text{ and} \\ l \text{ does not occur in } L \text{ and} \\ \text{a rule in } g(\Pi) \cup \Gamma_{\mathcal{L}} \text{ is equivalent to} \\ C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$$

Right Rules

$Backjump_{\mathcal{R}}$:

$$(L, Rl^{\Delta}R')_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L\bar{l}, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} Rl^{\Delta}R' \text{ is inconsistent and} \\ t(\Pi, L) \models \bar{l} \vee \overline{R} \end{cases}$$

$Learn_{\mathcal{R}}$:

$$(L, \emptyset) \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow L \| (\Gamma_{\mathcal{L}}, C \cup \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in } t(\Pi, L) \text{ and} \\ g(\Pi) \models C \end{cases}$$

$UnitDPL_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L, Rl)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \text{ if} \begin{cases} l \text{ is a literal over } atoms(t(\Pi, L)) \text{ and} \\ l \text{ does not occur in } R \text{ and} \\ \text{a rule in } t(\Pi, L) \cup \Gamma_{\mathcal{R}} \text{ is equivalent to} \\ C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } R \end{cases}$$

Crossing Rules

$Backjump_{\mathcal{RL}}$:

$$(Ll^{\Delta}L', R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L\bar{l}, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \quad \text{if} \begin{cases} \text{no right-rule applies and} \\ g(\Pi) \models \bar{l} \vee \overline{L} \end{cases}$$

Figure 5.1: New rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$.

As before, the added transition rules can be divided into one of the groups *Left rules*, *Right rules* and *Crossing Rules*, where the left rules capture the generate layer and the right rules capture the test layer. The crossing rules capture the transition from one layer to the other.

The transition rules $Backjump_{\mathcal{L}}$, $Unit_{\mathcal{L}}$, $Backjump_{\mathcal{R}}$ and $Unit_{\mathcal{R}}$ do not change the

learned clauses when being applied. The rules $Backjump_{\mathcal{L}}$ and $Backjump_{\mathcal{R}}$ are being used instead of $Backtrack_{\mathcal{L}}$ and $Backtrack_{\mathcal{R}}$, respectively, to reverse not only the last made decision, but also to skip some decisions that have nothing to do with the inconsistency. The rules $Unit_{\mathcal{L}}$ ($Unit_{\mathcal{R}}$) not only consider the rules of the generate (witness) layer, but also the learned clauses that are being captured in $\Gamma_{\mathcal{L}}$ ($\Gamma_{\mathcal{R}}$).

To learn a clause for the generate layer $Learn_{\mathcal{L}}$ can be used, while $Learn_{\mathcal{R}}$ is being used to add clauses to the witness layer.

The transition rule $Backjump_{\mathcal{RL}}$ is being used instead of $Backtrack_{\mathcal{RL}}$, possibly to undo multiple decisions at once when there are more than one decision literals in the generate layer. If it cannot be applied, it follows that no other candidate L can be found by the generate layer, so the transition $Conclude_{\mathcal{RL}}$ leading to $Failstate$ is inevitable.

The reason for $\Gamma_{\mathcal{R}} = \emptyset$ on each left rule, as well as on $Backjump_{\mathcal{RL}}$ is, because $\Gamma_{\mathcal{R}}$ are the learned clauses for the witness layer $t(\Pi, L)$, which is being dependent on the model L. As the model L changes, the rules of $t(\Pi, L)$ change. When it changes we cannot guarantee that the learned clauses from the previous computations are still valid, when we apply a left rule.

As we want to build a theorem about the features of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ like in the previous chapter, we have to take a closer look at finiteness. A change to some of the previous papers is that we work with sets instead of multisets (a set, where elements do not have to be unique) for $\Gamma_{\mathcal{L}}$ and $\Gamma_{\mathcal{R}}$. In the case of using multisets we would lose finiteness when we would learn the same clause over and over again.

Now, the transition rules $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ as they are stated at this moment could be applied over and over again as well, because there is no restriction that implies otherwise. The only difference is that the node of the graph doesn't change when we learn the same clause over and over again. Therefore, the graph is not acyclic, as learning the same clause twice in a row leads to a cycle from one node to the same node, as learning the same rule will not change the set of learned rules.

Theorem 2. *For a disjunctive program Π, a generating function g and a witness function t there holds*

1. *the graph $DPL^2_{g,t}(\Pi)$ is finite if there are no infinite subderivations of only Learn steps.*
2. *Any terminal state in $DPL^2_{g,t}(\Pi)$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L being a model of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if $g(\Pi)$ has no model such that its witness is unsatisfiable.*

Proof. **1.** In order to show that the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ is finite, consider some state $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ of $\mathrm{DPL}^2_{g,t}(\Pi)$. The rules that change the state $(L, R)_s$, but not the clauses $(\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ coincide with the ones in $\mathrm{DP}^2_{g,t}(\Pi)$. Using only these rules, the amount of possible states is finite, as it was proven in [BLM16]: Observing a string L of some state or of $Ok(L)$ we recognize multiple things:

1. L is built over a set of atoms that is bounded by the size of Π.

2. L does not allow repetitions.

It follows that there is a finite number of possible strings L in $(L, R)_s$ and $Ok(L)$. If there is only a finite number of L there is also a finite number of states $Ok(L)$ in our graph. Observing a string R of some state we see that because $t(\Pi, L)$ has a finite number of atoms and there are finitely many L, the set of atoms over which R is built is finite. It follows that there are finitely many possibilities for R. In total, there are finitely many states $(L, R)_s$. The rules $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ could lead to infiniteness when applied over and over again, but it is explicitely excluded in the theorem, as there are no infinite occurrences of Learn steps.

2. For the second part, we recall that there are three states: $Ok(L)$, $Failstate$ and a pair $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ with s being either $\mathcal{L}$ or $\mathcal{R}$. We consider the state $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ and show, that this is not a terminal state.
Case 1: Assume, that $s = \mathcal{L}$. In this case either a left-rule or $Cross_{\mathcal{LR}}$ applies. If none of the left-rule applies, $Cross_{\mathcal{LR}}$ applies in any case, as the only condition for the application of the rule is, that no left-rule applies. Thus, $(L, R)_{\mathcal{L}}$ is not a terminal state, as there is always a rule, we can apply.
Case 2: Assume, that $s = \mathcal{R}$. Neither $Conclude_{\mathcal{R}}$ nor $Conclude_{\mathcal{RL}}$ apply, as we want a state of the form $(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ to be a terminal state. Either a right-rule will be applied, which will lead us to a new state of the form $(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$, or the crossing-rule $Backjump_{\mathcal{RL}}$ holds, which would mean that L contains a decision literal. That rule leads to an state $(L, R)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$, where we already have shown, that this is not terminal.

Recall that a terminal state is a state where a transition rule leads to it, but none leave from it. Assume that $Failstate$ is reachable from the initial state. Following the applied transition rules that lead to $Failstate$ either $Conclude_{\mathcal{L}}$ or $Conclude_{\mathcal{RL}}$ was applied. There are no transition rules that origin at $Failstate$. Therefore $Failstate$ is a terminal state.

Assume that $Ok(L)$ is reachable from the initial state. There is a transition leading to this state, as it is different from the initial state. The ingoing transition rule can only be $Conclude_{\mathcal{R}}$. By definition, it holds that R is inconsistent and does not contain any decision literals. Thus, we already know that the program used for computing R, $t(\Pi, L)$, is unsatisfiable.

Looking at the rules, we know that $Ok(L)$ is achieved by applying a right-rule. As the initial state is $(\emptyset, \emptyset)_{\mathcal{L}}$, the crossing-rule $\mathcal{LR}$ was applied earlier, as it is the only way to reach the label $\mathcal{R}$. The rule $Cross_{\mathcal{LR}}$ is applied, when none of the left-rules hold. Assume that L is not a model. Then L is either not complete, but consistent or inconsistent. Assume that L is not complete, but consistent. If none of the left-rules $Backtrack_{\mathcal{L}}$, $UnitDPL_{\mathcal{L}}$ or $Learn_{\mathcal{L}}$ hold, we can always apply $Decide_{\mathcal{L}}$, as L is consistent and there is a literal in $atoms(g(\Pi))$ that does not occur in L, by the definition of completeness and consistency. If we can apply $Decide_{\mathcal{L}}$, we cannot apply $Cross_{\mathcal{LR}}$, which is needed to reach $Ok(L)$

If L is inconsistent and it contains no decision literals then $Failstate$ is being reached, which is a terminal state. Thus, the rule $Cross_{\mathcal{LR}}$ cannot be reached as well. If it contains decision literals, the rule $Backjump_{\mathcal{L}}$ is being applied which results in an L that is either not complete or inconsistent. There we have shown that the rule $Cross_{\mathcal{LR}}$ cannot be reached. Thus, if L is not a model, $Ok(L)$ cannot be reached.

3. For the third part, first we want to prove that Failstate is reachable if F has no model such that its witness is unsatisfiable.

Assume that $Failstate$ is not reachable. From (2) we know that there are the terminal states $Failstate$ and $Ok(L)$. As we need to reach a terminal state at some point, the only terminal state possible is $Ok(L)$. The only way to reach $Ok(L)$ is by applying the right-rule $Conclude_{\mathcal{R}}$. Since F has no model such that its witness is unsatisfiable we cannot apply $Conclude_{\mathcal{R}}$. Thus, there is no way to get to $Ok(L)$. We derive a contradiction.

Now we show that if $Failstate$ is reachable, then there is no model of F such that the witness is unsatisfiable. Assume that there is a model of F such that the witness is unsatisfiable. Then, we can apply $Conclude_{\mathcal{R}}$, which will lead us to $Ok(L)$. But then $Failstate$ is not reachable and we derive a contradiction. □

To make the graph finite, we can add some restriction like we did in Theorem 2. Using the constraint of not having infinite subdervations of learn steps will not give us acyclicity, as it is not forbidden to learn the same rule at least twice in a row which would lead to a cycle on one node. What we can do to achieve both, finiteness and acyclicity, is to adapt the transition rules $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ in Figure 5.1 and exchange them with $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ from Figure 5.2.

For the rest of the paper, when we will talk about $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ the form from Figure 5.2 will be referred to except stated otherwise. When talking about $\mathrm{DPL}^2_{g,t}(\Pi)$ we will mean the graph with the learn rules from Figure 5.2.

Left Rules

$Learn_{\mathcal{L}}$:

$$(L,\emptyset)\|(\Gamma_{\mathcal{L}},\emptyset) \implies (L,\emptyset)\|(C\cup\Gamma_{\mathcal{L}},\emptyset) \text{ if } \begin{cases} \text{every atom in } C \text{ occurs in } g(\Pi) \text{ and} \\ g(\Pi)\models C \text{ and} \\ C\notin\Gamma_{\mathcal{L}} \end{cases}$$

Right Rules

$Learn_{\mathcal{R}}$:

$$(L,\emptyset)\|(\Gamma_{\mathcal{L}},\Gamma_{\mathcal{R}}) \implies L\|(\Gamma_{\mathcal{L}},C\cup\Gamma_{\mathcal{R}}) \quad \text{if } \begin{cases} \text{every atom in } C \text{ occurs in } t(\Pi,L) \text{ and} \\ g(\Pi)\models C \text{ and} \\ C\notin\Gamma_{\mathcal{L}} \end{cases}$$

Figure 5.2: Adapted learn rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$.

The following theorem is similar to Theorem 2 with the difference of using the learning rules from Figure 5.2:

Theorem 3. *For a disjunctive program* Π*, a generating function* g *and a witness function* t *there holds*

1. *the graph* $DPL^2_{g,t}(\Pi)$ *is finite and acyclic.*

2. *Any terminal state in* $DPL^2_{g,t}(\Pi)$ *reachable from the initial state is either* $Failstate$ *or of the form* $Ok(L)$*, with* L *being a model of* $g(\Pi)$ *such that* $t(\Pi, L)$ *is unsatisfiable.*

3. $Failstate$ *is reachable from the initial state if and only if* $g(\Pi)$ *has no model such that its witness is unsatisfiable.*

Proof. **1.** The proof of finiteness follows the lines of Theorem 2 up to the line "... In total, there are finitely many states $(L, R)_s$." Now we have to have a look at the Learn rules $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ differently. The two learning rules change either $\Gamma_{\mathcal{L}}$ or $\Gamma_{\mathcal{R}}$, which are sets of clauses over atoms of $g(\Pi)$. As $g(\Pi)$ consists of finitely many atoms, there are finitely many possibilities to put these atoms together to receive learning clauses. As a set contains each possible clause only once and for each clause that we can learn we can apply the according transition rule only once, we remain finite.

To show acyclicity, we write a string L of literals as $L_0 l_1^{\Delta} L_1 \dots l_k^{\Delta} L_k$, where $(l_t^{\Delta})_{1 \leq t \leq k}$ are all the decision literals of L. $|L|$ refers to the length of L and $v(L)$ is the sequence $|L_0|, |L_1|, \dots |L_k|$. For a set of clauses Γ, we define $v(\Gamma)$ as the sum of numbers of atoms in Γ, for example if $\Gamma = \{a \cup b, a \cup c\}$, then $v(\Gamma) = 4$.

$L \leq L'$ if and only if $v(L) \leq_{lex} v(L')$, where $\leq_{lex}$ is the lexicographic order. In this case, the lexicographic order means, if we have two sequences $a = (a_1, a_2, \dots)$ and $b = (b_1, b_2, \dots)$ then $a \leq b$ if the first i where a_i and b_i differ holds $a_i \leq b_i \cap b_i \not\leq a_i$ or if a is a subset of b, where $a_i = b_i$ for each $i \in a$.

The length of the sequence $v(L)$ is bounded, because of the finiteness of possible decision literals, we got a well-founded order. There holds $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$ if and only if $v(L, R, s, \Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq_{lex} v(L', R', s', \Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$, with $\mathcal{L} \leq \mathcal{R}$. This is well-founded as it is the lexicographic composition of well-founded orders.

For any transition $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ to $(L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$ there holds $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$ and $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \neq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$.

$Backjump_{\mathcal{L}}$ has the form $(L_0 l_1^{\Delta} L_1, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \Longrightarrow (L'_0, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset)$, where $L_0 \leq L'_0 = |L_0| + 1 + |L_1|$. Therefore, $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$.
$Learn_{\mathcal{L}}$ has the form $(L, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \Longrightarrow (L, \emptyset)_{\mathcal{L}} \| (C \cup \Gamma_{\mathcal{L}}, \emptyset)$, where everything up until $\Gamma_{\mathcal{L}}$ is equal and $|\Gamma_{\mathcal{L}}| \leq |\Gamma_{\mathcal{L}}| + |C|$. There holds $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$. Applying $Unit_{\mathcal{L}}$ we quickly see that $|L'| = |L| + 1$ and therefore, $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq (L', R')_{s'} \| (\Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$. The right and crossing rules work analogously to the left rules.

Because the order is well-founded, there is no infinite path in the graph. Therefore, the graph is acyclic.

2. + 3. These parts coincede with the proof of Theorem 2. □

5.1.1 Extension of $DPL^2_{g,t}(\Pi)$ to Forgetting

As we have extended $DPL^2_{g,t}(\Pi)$ to a Learning rule, it makes sense to consider the opposite of the rule, as we have done in Chapter 2.2.3. Therefore, we introduce the transition rules $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$ to delete some of the learned rules in $\Gamma_{\mathcal{L}}$ or $\Gamma_{\mathcal{R}}$, respectively. The rule $Forget_{\mathcal{L}}$ ($Forget_{\mathcal{R}}$) is added to the left-rules (right-rules) of $DPL^2_{g,t}(\Pi)$. A clause C can be forgotten, when each atom in that clause occurs in $g(\Pi) \cup \Gamma_{\mathcal{L}}$ ($t(\Pi, L) \cup \Gamma_{\mathcal{R}}$) and

$g(\Pi)$ ($t(\Pi, L)$) entails $\Gamma_{\mathcal{L}}$ ($\Gamma_{\mathcal{R}}$).

Left-rules
$Forget_{\mathcal{L}}$:

$$(L, \emptyset)_{\mathcal{L}} \| (C \cup \Gamma_{\mathcal{L}}, \emptyset) \implies (L, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in} \\ g(\Pi) \text{ and} \\ g(\Pi) \models C \end{cases}$$

Right-rules
$Forget_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, C \cup \Gamma_{\mathcal{R}}) \implies (L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \text{ if } \begin{cases} \text{every atom in } C \text{ occurs in} \\ t(\Pi, L) \text{ and} \\ t(\Pi, L) \models C \end{cases}$$

Figure 5.3: Forgetting rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$.

Theorem 4. *For a disjunctive program Π, a generating function g and a witness function t there holds*

1. *graph $DPL^2_{g,t}(\Pi)$ is finite, if it does not contain an infinite subsequence of only Learn and Forget steps.*
2. *Any terminal state in $DPL^2_{g,t}(\Pi)$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L being a model of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if $g(\Pi)$ has no model such that its witness is unsatisfiable.*

Proof. The case of finiteness under the condition that there is no clause, that is learned and then forgotten for infinitely many times, leads to the same setting as in Theorem 3, where we have showed that the graph remains finite.

The proof of the second part follows the lines of Theorem 3. The rules $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$ do not change the terminal states, as applying one of these transition rules to a state of the form $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ will lead to a state of the same form, where we have shown that these will lead to a terminal state that is either $Failstate$ or $Ok(L)$. This will lead to the second part of the proposition.

For the third part we see that the transition rule $Forget$ does not lead to $Failstate$. Therefore the proof is equal to the proof of Theorem 3. □

Remark 13. Note that using Learning and Forgetting rules may lead to graphs that are not acyclic, as we can learn and forget the same rule over and over again.

5.1.2 Extension of $\mathrm{DPL}^2_{g,t}(\Pi)$ to Restart

As seen in Chapter 2.2.3, adding the possibility of restarting the search for an answer set while keeping the clauses that have been learned to that point, can lead to a more

efficient computation of finding a model. For this reason, we introduce the transition rules $Restart_{\mathcal{L}}$ and $Restart_{\mathcal{R}}$ for $\mathrm{DPL}^2_{g,t}(\Pi)$ as seen in Figure 5.4. There is no condition needed, as theoretically we could apply the restart rules at any time. When applying a Restart rule we keep the learned clauses while resetting either L on applying $Restart_{\mathcal{L}}$ or R on applying $Restart_{\mathcal{R}}$ in hope of a more efficient computation after the Restart.

Left-rules
$Restart_{\mathcal{L}}$:
$(L, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset) \quad \Longrightarrow (\emptyset, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset)$

Right-rules
$Restart_{\mathcal{R}}$:
$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L, \emptyset)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$

Figure 5.4: Restart rules of the graph $\mathrm{DPL}^2_{g,t}(\Pi)$.

Theorem 5. *For a disjunctive program Π, a generating function g and a witness function t there holds*

1. *graph $DPL^2_{g,t}(\Pi)$ is finite, if it does not contain an infinite subsequence of only Learn and Forget steps and Restart has increasing periodicity in it.*
2. *Any terminal state in $DPL^2_{g,t}(\Pi)$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L being a model of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if $g(\Pi))$ has no model such that its witness is unsatisfiable.*

Proof. Assume there are infinitely many occurrences of $Restart_s$, where s is a Label. It is already known from Theorem 4, that the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ with *Learn* and *Forget* is finite under the constraint of not having infinitely many Learn and Forget steps. Assume that the number of possible states of the form $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ is n. Applying a transition rule that is not restart on such a state will lead to a different state. There cannot be more than n steps between applying *Restart* at the beginning and at the end, where it does not matter whether we apply $Restart_{\mathcal{L}}$ or $Restart_{\mathcal{R}}$. So, there cannot be infinitely many Restarts if Restart has increasing periodicity.

The proof of the second and the third part follows the lines of Theorem 4. There, the transition rule *Restart* can be handled like the transition rule *Forget*. □

5.2 Extending GNT to Backjumping and Learning

In Chapter 4.3, we reviewed the graph $SM^2_{g(\Pi),t}$. Now we will extend it to backjumping and learning to reach the graph $SML^2_{g(\Pi),t}$. As before, $g(\Pi)$ is the generating function and $t(\Pi, L)$ is the witness function. To capture the rule Learn, we need to add the learned clauses to the state, so we will use the extended state from Definition 46 in the graph $SML^2_{g(\Pi),t}$ as state. We will also take a look at the features of the program, but we will not prove the statements in this section, because the statement will only be a special case of the proof in Chapter 6. The extension of the graph $SM^2_{g(\Pi),t}$ is similar to the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ in CMODELS.

In Figure 5.6, 5.5 and 5.7 we present the rules of abstract GNT with backjumping and learning. The rules are a combination of the rules that we have seen in CMODELS as well as in abstract GNT with backtracking, where we extend the rules as seen in Equation 5.1, 5.2, 5.3 and 5.4.

We see that for extending to backjumping we will basically replace the rule *Backtrack* with *Backjump* and *Unit* with *UnitDPL*, like in CMODELS.

To extend the graph to the transition rule Learn, we can just use the rules $Learn_{\mathcal{L}}$ and $Learn_{\mathcal{R}}$ from $\mathrm{DPL}^2_{g,t}(\Pi)$ in Figure 5.2.

Similar to before, we can build the graph $SML^2_{g(\Pi),t}$ by using the states as nodes and the transition rules from Figure 5.6, 5.5 and 5.7 as edges.

The following theorem will be proven in Chapter 6, as the general framework that we will talk about will capture this solver. It is similar to Theorem 3.

Proposition 8. *[BLM14] For a disjunctive program Π, a generating function g and a witness function t there holds*

1. *graph $SML^2_{g(\Pi),t}$ is finite and acyclic.*
2. *Any terminal state of $SML^2_{g(\Pi),t}$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L^+ being an answer set (stable model) of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if there is no set L of literals such that L^+ is an answer set of $g(\Pi)$ such that its witness is unsatisfiable.*

Example 18. Assume a program Π

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

Applying the generating function $g^G(\Pi)$ from Equation 4.4 to the program leads to the following generate layer:

$$\begin{aligned} g^G(\Pi) = & a \leftarrow \neg a^r \\ & b \leftarrow \neg b^r \\ & a^r \leftarrow \neg a \\ & b^r \leftarrow \neg b \\ & \leftarrow \neg a, \neg b \\ & a \leftarrow c \\ & c \leftarrow \\ & d \leftarrow e \\ & a^s \leftarrow \neg b \\ & b^s \leftarrow \neg a \\ & \leftarrow a, \neg a^s \\ & \leftarrow b, \neg b^s. \end{aligned}$$

A possible path could look as the following:

$$\begin{aligned} & (\emptyset, \emptyset)_{\mathcal{L}} && \Rightarrow && (Decide_{\mathcal{L}}) \\ & (\neg a^{\Delta}, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r \neg b, \emptyset)_{\mathcal{L}} && \Rightarrow && (Decide_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r \neg b d^{\Delta}, \emptyset)_{\mathcal{L}} && \Rightarrow && (BackchainTrue_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r \neg b d^{\Delta} e, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r \neg b d^{\Delta} e c, \emptyset)_{\mathcal{L}} && \Rightarrow && (BackchainTrue_{\mathcal{L}}) \\ & (\neg a^{\Delta} a^r \neg b d^{\Delta} e c \neg c, \emptyset)_{\mathcal{L}} && \Rightarrow && (Backjump_{\mathcal{L}}) \\ & (a, \emptyset)_{\mathcal{L}} && \Rightarrow && (BackchainTrue_{\mathcal{L}}) \\ & (ac, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (ac \neg a^r, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (ac \neg a^r \neg b, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (ac \neg a^r \neg b b^r, \emptyset)_{\mathcal{L}} && \Rightarrow && (AllRulesCancelled_{\mathcal{L}}) \\ & (ac \neg a^r \neg b b^r \neg d, \emptyset)_{\mathcal{L}} && \Rightarrow && (BackchainTrue_{\mathcal{L}}) \\ & (ac \neg a^r \neg b b^r \neg d \neg e, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (ac \neg a^r \neg b b^r \neg d \neg e a^s, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \\ & (ac \neg a^r \neg b b^r \neg d \neg e a^s \neg b^s, \emptyset)_{\mathcal{L}} && \Rightarrow && (UnitDPL_{\mathcal{L}}) \end{aligned}$$

On applying transition rules from Figure 5.2 to the generate layer, we decide that a is negative which leads us to multiple applications of the transition rule $UnitDPL_{\mathcal{L}}$. After deciding on the literal d we continue applying $BackchainTrue_{\mathcal{L}}$ and $UnitDPL_{\mathcal{L}}$ until we

run into an inconsistent state. The reason for this seems to be the literal a and not d. In the version from [BLM16] we would change the assignment of d first, reach another inconsistent state and then change the assignment of a. Now, we can skip this middle part, go right away to the source of the problem and change the assignment of a. Applying further transition rules will lead to a consistent state

$$L = ac\neg a^r\neg bb^r dea^s\neg b^s.$$

Now, we can look at the test layer. Applying Equation 4.5 leads to the program

$$\begin{aligned} t^G(\Pi) = \quad & a \leftarrow \neg a^r \\ & a^r \leftarrow \neg a \\ & \neg a, \neg b \\ & a \leftarrow c \\ & c \leftarrow \\ & \leftarrow a\neg bc\neg d\neg e. \end{aligned} \tag{5.5}$$

$$\begin{aligned} & (L, \emptyset)_{\mathcal{L}} && \Rightarrow && (Cross_{\mathcal{LR}}) \\ & (L, \emptyset)_{\mathcal{R}} && \Rightarrow && (UnitDPL_{\mathcal{R}}) \\ & (L, c)_{\mathcal{R}} && \Rightarrow && (UnitDPL_{\mathcal{R}}) \\ & (L, ca)_{\mathcal{R}} && \Rightarrow && (AllRulesCancelled_{\mathcal{R}}) \\ & (L, ca\neg d)_{\mathcal{R}} && \Rightarrow && (AllRulesCancelled_{\mathcal{R}}) \\ & (L, ca\neg d\neg e)_{\mathcal{R}} && \Rightarrow && (BackchainTrue_{\mathcal{R}}) \\ & (L, ca\neg d\neg ea^r)_{\mathcal{R}} && \Rightarrow && (AllRulesCancelled_{\mathcal{R}}) \\ & (L, ca\neg d\neg ea^r\neg b)_{\mathcal{R}} && \Rightarrow && Conclude_{\mathcal{R}} \\ & Ok(L) \end{aligned}$$

Now, as the test layer is definded in Equation 5.5, we can apply a crossing rule to go to the test layer to check for minimality. Applying multiple transition rules leads to an inconsistent state, as the rule

$$\leftarrow a\neg bc\neg d\neg e$$

is not being fulfilled and there are no decision literals. Therefore, we reach the final state $Ok(L)$ with the set $\{a, c\}$ being an answer set of $g(\Pi)$ where $t(\Pi, L)$ is unsatisfiable.

Left Rules

$Conclude_{\mathcal{L}}$:

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow Failstate \quad \text{if} \begin{cases} L \text{ is inconsistent and} \\ L \text{ contains no decision literal} \end{cases}$$

$Decide_{\mathcal{L}}$

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (Ll^{\Delta},\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} L \text{ is consistent and} \\ l \text{ is a literal over } atoms(g(\Pi)) \text{ and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } L \end{cases}$$

$Backjump_{\mathcal{L}}$:

$$(Ll^{\Delta}L',\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (L\bar{l},\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} Ll^{\Delta}L' \text{ is inconsistent and} \\ g(\Pi) \models l' \vee \overline{L} \end{cases}$$

$Learn_{\mathcal{L}}$:

$$(L,\emptyset) \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (L,\emptyset) \| (C \cup \Gamma_{\mathcal{L}},\emptyset) \text{ if } \begin{cases} \text{every atom in } C \text{ occurs in } g(\Pi) \text{ and} \\ g(\Pi) \models C \text{ and} \\ C \notin \Gamma_{\mathcal{L}} \end{cases}$$

$UnitDPL_{\mathcal{L}}$:

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (Ll,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} l \text{ is a literal over } atoms(t(\Pi,L)) \text{ and} \\ l \text{ does not occur in } R \text{ and} \\ \text{a rule in } t(\Pi,L) \cup \Gamma_{\mathcal{L}} \text{ is equivalent to} \\ C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$$

$AllRulesCancelled_{\mathcal{L}}$:

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (L\bar{l},\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} \text{there is no rule in } g(\Pi) \text{ supporting } l \\ \text{with respect to } L \end{cases}$$

$BackchainTrue_{\mathcal{L}}$:

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (Ll,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} \text{there is a rule } A \vee a \leftarrow B \text{ in } g(\Pi) \\ \text{so that (i) } a \in L \text{ and} \\ \text{(ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and} \\ \text{(iii) no other rule in } g(\Pi) \text{ is supporting a} \\ \text{with respect to } L \end{cases}$$

$Unfounded_{\mathcal{L}}$:

$$(L,\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \Longrightarrow (L\bar{l},\emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}},\emptyset) \quad \text{if} \begin{cases} L \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } l \text{ such that} \\ X \text{ is unfounded on } L \text{ with respect to } g(\Pi) \end{cases}$$

Figure 5.5: Left rules of the graph $SML^2_{g(\Pi),t}$.

Right Rules

$Conclude_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow Ok(L) \quad \text{if} \begin{cases} R \text{ is inconsistent and} \\ R \text{ contains no decision literal} \end{cases}$$

$Decide_{\mathcal{R}}$

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow (L, Rl^{\Delta})_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \text{ if} \begin{cases} R \text{ is consistent and} \\ l \text{ is a literal over } atoms(t(\Pi, L)) \text{ and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } R \end{cases}$$

$Backjump_{\mathcal{R}}$:

$$(L, Rl^{\Delta}R')_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \Longrightarrow (L\bar{l}, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} Rl^{\Delta}R' \text{ is inconsistent and} \\ t(\Pi, L) \models l' \vee \overline{R} \end{cases}$$

$Learn_{\mathcal{R}}$:

$$(L, \emptyset) \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow L \| (\Gamma_{\mathcal{L}}, C \cup \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in } t(\Pi, L) \text{ and} \\ g(\Pi) \models C \text{ and} \\ C \notin \Gamma_{\mathcal{L}} \end{cases}$$

$UnitDPL_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow (L, Rl)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} l \text{ is a literal over } atoms(t(\Pi, L)) \text{ and} \\ l \text{ does not occur in } R \text{ and} \\ \text{a rule in } t(\Pi, L) \cup \Gamma_{\mathcal{R}} \text{ is equivalent to} \\ C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } R \end{cases}$$

$AllRulesCancelled_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow (L, R\bar{l})_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} \text{there is no rule in } t(\Pi, L) \text{ supporting } l \\ \text{with respect to } R \end{cases}$$

$BackchainTrue_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow (L, Rl)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} \text{there is a rule } A \vee a \leftarrow B \text{ in } t(\Pi, L) \\ \text{so that (i) } a \in R \text{ and} \\ \text{(ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and} \\ \text{(iii) no other rule in } t(\Pi, L) \text{ is supporting } a \\ \text{with respect to } R \end{cases}$$

$Unfounded_{\mathcal{R}}$:

$$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \Longrightarrow (L, R\bar{l})_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \quad \text{if} \begin{cases} R \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } l \\ \text{such that } X \text{ is unfounded on } R \\ \text{with respect to } t(\Pi, L) \end{cases}$$

Figure 5.6: Right rules of the graph $SML^2_{g(\Pi),t}$.

Crossing-rule $\mathcal{LR}$
$Cross_{\mathcal{LR}}$:
$(L, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \implies (L, \emptyset)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ if { no left-rule applies

Crossing-rules $\mathcal{RL}$
$Conclude_{\mathcal{RL}}$:
$(L, R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \implies Failstate$ if { no right-rule applies and / L contains no decision literal

$Backjump_{\mathcal{RL}}$:
$(Ll^{\Delta} L', R)_{\mathcal{R}} \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \implies (L\bar{l}, \emptyset)_{\mathcal{L}} \| (\Gamma_{\mathcal{L}}, \emptyset)$ if { no right-rule applies and / $g(\Pi) \models l' \vee \overline{L}$

Figure 5.7: The crossing rules of the graph $SML^2_{g(\Pi),t}$.

When extending the graph $SML^2_{g(\Pi),t}$ with the rules $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$, which we can take from $\mathrm{DPL}^2_{g,t}(\Pi)$ as well, we need to remember that we will have to restrict the graph to a finite amount of *Learn* and *Forget* steps to remain finite. We find these rules in Figure 5.3. The following propositions are similar to Theorem 4 and Theorem 5.

Proposition 9. *Given a program Π, a generating function g and a witness function t, there holds*

1. *graph $SML^2_{g(\Pi),t}$ with Forgetting is finite, if it does not contain an infinite subsequence of only Learn and Forget steps.*
2. *Any terminal state in $SML^2_{g(\Pi),t}$ with Forgetting reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L^+ being an answer set of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if there is no set L of literals such that L^+ is an answer set of $g(\Pi)$ such that its witness is unsatisfiable.*

The extension to Restart is similar to the Extension of $\mathrm{DPL}^2_{g,t}(\Pi)$ to Restart in Figure 5.4 as well.

Proposition 10. *Given a program Π, a generating function g and a witness function t, there holds*

1. *graph $SML^2_{g(\Pi),t}$ with Forgetting and Restarting is finite, if it does not contain an infinite subsequence of only Learn and Forget steps and Restart has increasing periodicity in it.*
2. *Any terminal state in $SML^2_{g(\Pi),t}$ with Forgetting and Restarting reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with L^+ being a answer set of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable.*
3. *$Failstate$ is reachable from the initial state if and only if there is no set L of literals such that L^+ is an answer set of $g(\Pi)$ such that its witness is unsatisfiable.*

5.3 Extending DLV to Backjumping and Learning

Again, we want to extend the graph $(SM^* \times DP)_{g(\Pi),t}$ to backjumping and learning, as well as to forgetting and restarting. We proceed as we did for GNT and CMODELS in Section 5.2 by remembering learned clauses. We already know that the abstract approach to DLV leads to a subset of GNT. So this time we will not state the rules again, but just captures the extension.

We use the given generating function $g(\Pi)$ as well as the witness function $t(\Pi, L)$ and the extended state from Definition 46. As we have seen in Chapter 4.4 the generating function for DLV is the identity function, so we can omit this part in the theorem.

To extend the graph to Backjumping, we will replace *Backtrack* with *Backjump* for both layers.

To capture the transition rule *Learn*, we add the same rules as in Figure 5.2.

This will lead us to the following propositions:

Proposition 11. *[BLM14] For a disjunctive program* Π *and a witness function* t *there holds*

1. *graph* $(SML^* \times DPL)_{g(\Pi),t}$ *is finite and acyclic.*
2. *Any terminal state of* $(SML^* \times DPL)_{g(\Pi),t}$ *reachable from the initial state and other than* $Failstate$ *is* $Ok(L)$*, with* L^+ *being an answer set (stable model) of* Π *such that* $t(\Pi, L)$ *is unsatisfiable.*
3. $Failstate$ *is reachable from the initial state if and only if there is no set* L *of literals such that* L^+ *is an answer set of* Π *such that its witness is unsatisfiable.*

Extending the graph $(SML^* \times DPL)_{g(\Pi),t}$ with the rules *Forget* can be realized by adding $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$ from Figure 5.3.

Proposition 12. *Given a program* Π *and a witness function* t *there holds*

1. *graph* $(SML^* \times DPL)_{g(\Pi),t}$ *with Forgetting is finite, if it does not contain an infinite subsequence of only Learn and Forget steps.*
2. *Any terminal state in* $(SML^* \times DPL)_{g(\Pi),t}$ *with Forgetting reachable from the initial state is either* $Failstate$ *or of the form* $Ok(L)$*, with* L^+ *being an answer set of* Π *such that* $t(\Pi, L)$ *is unsatisfiable.*
3. $Failstate$ *is reachable from the initial state if and only if there is no set* L *of literals such that* L^+ *is an answer set of* Π *such that its witness is unsatisfiable.*

The rule *Restart* is the same as in CMODELS, which can be found in Figure 5.4.

Proposition 13. *Given a program* Π *and a witness function* t*, there holds*

1. *graph* $(SML^* \times DPL)_{g(\Pi),t}$ *with Forgetting and Restarting is finite, if it does not contain an infinite subsequence of only Learn and Forget steps and Restart has increasing periodicity in it.*

2. *Any terminal state in* $(SML^* \times DPL)_{g(\Pi),t}$ *with Forgetting and Restarting reachable from the initial state is either* $Failstate$ *or of the form* $Ok(L)$*, with* L^+ *being a answer set of* Π *such that* $t(\Pi, L)$ *is unsatisfiable.*

3. $Failstate$ *is reachable from the initial state if and only if there is no set* L *of literals such that* L^+ *is an answer set of* Π *such that its witness is unsatisfiable.*

6 Graph Template

We have seen that abstract CMODELS, abstract GNT and abstract DLV are similar in a sense that they are based on a two-layer approach and perform the task of computing answer sets of logic programs. In [BLM16], an approach for having a unifying framework for capturing two-layer methods was being made by introducing a graph template to encompass these disjunctive solvers. The authors proved the same features as we have seen in the earlier chapters, namely finiteness, termination, correctness and acyclicity for this general framework. In this work, we will extend the earlier to backjumping, learning, forgetting and restart. The special thing about the framework is that it can be used as a tool for designing new algorithms by combining different techniques of different solvers.

The template uses so-called propagator conditions to capture the transition rules.

First, we will look at the single layer graph template according to [BLM16] to introduce the notations. Then we will add the proposed rules like *Backjump*, *Learn*, *Forget* and *Restart* to receive an extended form of the framework that will work for multiple solvers at once.

6.1 A Single Layer Graph Template

Using the notation of [BLM16], we will start with recreating the single layer graph template to use it for disjunctive answer set solvers with backjumping and learning.

Definition 47. A *propagator condition* (*p-condition*) is a function from a program Π and a set of literals over $atoms(\Pi)$ to a set of literals over $atoms(\Pi)$.

The four p-conditions needed for the template, namely $UnitPropagate$, $AllRulesCancelled$, $BackchainTrue$ and $Unfounded$, were being introduced in [BLM16]. They are shown in Figure 6.1.

Definition 48. Let Π be a program, M a set of literals and $\mathcal{P}$ a set of p-conditions. $\mathcal{P}(\Pi, M)$ is the set of literals $\bigcup_{p\in\mathcal{P}} p(\Pi, M)$. $\mathcal{P}(\Pi, M)$ can be seen as the union of the possible outcomes obtained from $\mathcal{P}$.

Definition 49. *[BLM16]* Given a program Π and a set $\mathcal{P}$ of p-conditions, the DPLL graph template $DPTL_{\mathcal{P},\Pi}$ is a graph that is defined as follows:

- The set of nodes are the basic states relative to $atoms(\Pi)$.
- The edges are specified by the transition rules *Conclude*, *Decide*, *Success* presented in Figure 2.3, the transition rules *Backjump* from Figure 2.7, *Learn* from Figure 2.8 and the transition rule

$$Propagate \quad L\|\Gamma \quad \Rightarrow \quad Ll\|\Gamma' \quad \text{if } l \in \mathcal{P}(\Pi, L). \tag{6.1}$$

$l \in UnitPropagate(\Pi, L)$
iff $\begin{cases} l \text{ does not occur in } L \text{ and} \\ \text{a rule in } \Pi \text{ that is equivalent to } C \vee l \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$

$\neg a \in AllRulesCancelled(\Pi, L)$
iff $\begin{cases} \neg a \text{ does not occur in } L \text{ and} \\ \text{there is no rule in } \Pi \text{ supporting } a \text{ with respect to } L \end{cases}$

$l \in BackchainTrue(\Pi, L)$
iff $\begin{cases} l \text{ does not occur in } L \text{ and} \\ \text{there is a rule } A \vee a \leftarrow B \text{ in } \Pi \\ \text{so that (i) } a \in L, \text{ and (ii) either } \bar{l} \in A \text{ or } \ l \in B \text{ and,} \\ \text{(iii) no other rule in } \Pi \text{ is supporting a with respect to } L \end{cases}$

$\neg a \in Unfounded(\Pi, L)$
iff $\begin{cases} \neg a \text{ does not occur in } L \text{ and} \\ L \text{ is consistent and} \\ \text{there is a set } X \text{ of atoms containing } a \text{ such that} \\ X \text{ is unfounded on } L \text{ with respect to } \Pi \end{cases}$

Figure 6.1: Propagator Conditions

Remark 14. The propagator conditions are basically a slightly different notation of those transition rules, that differ in the solvers CMODELS, GNT and DLV. We use this notation to capture different subsets of the propagator conditions for different solvers. For example, in CMODELS, we only need the propagator condition $UnitPropagate$. Therefore, the according instantiation is $\mathcal{P} = \{UnitPropagate\}$. We can write

$$DPTL_{\{UnitPropagate\},\Pi} = DPL_F.$$

Both share the same nodes, which are basic states relative to $atoms(\Pi)$. The transition rules that are not $Unit$ also coincide, as they are taken from the original definition. The last transition rule $Propagate$ is only applicable if $l \in UnitPropagate(\Pi, L)$ is applicable. The conditions coincide with the conditions of $Unit$. Therefore, in the case of $\mathcal{P} = \{UnitPropagate\}$, $Propagate$ coincides with $Unit$ in DPL_F.

Remark 15. Note that the transition rules $Backjump$, $Learn$, $Forget$ and $Restart$ are the same for CMODELS, GNT and DLV. Therefore, we do not need to capture them in the propagator conditions.

Remark 16. We will refer to the graph template with Forgetting as $DPTLF_{\mathcal{P},\Pi}$, where the additional edge of the graph $DPTL_{\mathcal{P},\Pi}$ is being captured by the transition rule *Forget* in Figure 2.10.

The graph template with Forgetting and Restarting will be named $DPT2_{\mathcal{P},\Pi}$. It will capture the graph $DPTLF_{\mathcal{P},\Pi}$ with the additional transition rule *Restart* in Figure 2.11.

Now, as the template is done, we want to review its instantiation as we want to capture different outcomes of a solver, like whether we compute a classical or a stable model. To do this, we introduce *types*.

Definition 50. By *type* we refer to an element of the set $T = \{cla, sup, sta\}$. A *cla*-model denotes a classical model, a *sup*-model a supported model and a *sta*-model a stable model.

Definition 51. Assume $\omega \in T$, where $T = \{cla, sup, sta\}$, Π be a program, M be a set of literals and M_1 be any ω-model of Π, such that $M \subset M_1$. A set $\mathcal{P}$ of p-conditions is *ω-sound* if $\mathcal{P}(\Pi, M) \subseteq M_1$.

Remark 17. Any *cla*-sound set of p-conditions is *sup*-sound. This is the case because every supported model is a classical model by definition.

Any *sup*-sound set of p-conditions is *sta*-sound, because every stable model is a supported model as well.

Definition 52. Let M be a consistent and complete set of literals over $atoms(\Pi)$, where Π is a program and $\mathcal{P}$ be a set of p-conditions. $\mathcal{P}$ is *ω-complete* when the set M is a ω-model of Π if and only if $\mathcal{P}(\Pi, M) = \emptyset$.

Definition 53. *A set $\mathcal{P}$ of p-conditions is ω-enforcing if $\mathcal{P}$ is ω-sound and ω-complete.*

Proposition 14. *[BLM16]* The following statements hold:

1. The set $\{UnitPropagate\}$ is *cla*-enforcing.
2. All the subsets of $\{UnitPropagate, AllRulesCancelled, BackchainTrue\}$ that contain $\{UnitPropagate, AllRulesCancelled\}$ are *sup*-enforcing.
3. All the subsets of $\{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}$ that contain $\{UnitPropagate, Unfounded\}$ are *sta*-enforcing.

Proof. see [BLM16] □

Proposition 15. *[BLM16]* For any program Π, type ω and a ω-enforcing set of p-conditions $\mathcal{P}$, there hold

1. the graph $DPTL_{\mathcal{P},\Pi}$ is finite, if there are no infinite subderivations of only Learn steps.
2. any terminal state of $DPTL_{\mathcal{P},\Pi}$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with $L_{|atoms(\Pi)}$ being a ω-model of Π.
3. $Failstate$ is reachable from the initial state if and only if there is no ω-model of Π.

Proof. The proof relies on the same proof techniques as the proof of Proposition 6. □

Remark 18. Note that when constructing the transition rule *Learn* like in Figure 2.8 we lose finiteness and acyclicity as there is no restriction of learning the same rule over and over again. To gain these features, we have to adapt the transition rule as in the two-layer structures, see Figure 5.2. We receive the transition rule in Figure 6.2.

$$\textit{Learn}: \qquad M\|\Gamma \Longrightarrow M\|C,\Gamma \quad \text{if} \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ and} \\ F \models C \text{ and} \\ C \notin \Gamma_{\mathcal{L}} \end{cases}$$

Figure 6.2: Adapted learn rule of DPL_F.

Remark 19. Here, the single layer graph template is used to understand the two-layer graph template, so we will not restate the propositions for $DPTL_{\mathcal{P},\Pi}$ with the adapted learn rule. It will add acyclicity to the proposition. The graph $DPTLF_{\mathcal{P},\Pi}$ with Backjumping, Learning and Forgetting will be finite when there are no infinite subsequences of only Learn and Forget steps. The graph $DPT2_{\mathcal{P},\Pi}$ with Backjumping, Learning, Forgetting and Restarting will only be finite if there holds both, no infinite subsequences of only Learn and Forget steps and increasing periodicity of Restart.

6.2 A Two-Layer Graph Template

As next step, we extend the two-layer graph template according to [BLM16] to Backjumping and Learning.

Definition 54. *[BLM16]* Let Π be a program, $\mathcal{P}_{\mathcal{L}}$ and $\mathcal{P}_{\mathcal{R}}$ be sets of p-conditions, g be a generating function and t be a witness function. The two-layer template graph $STTL^{\mathcal{P}_{\mathcal{L}},g}_{\mathcal{P}_{\mathcal{R}},t}$ is defined as follows:

1. The set of nodes are the set of states relative to $atoms(g(\Pi))$ and $atoms(t,\Pi,atoms(g(\Pi)))$.
2. The edges are the transition rules from graph $\mathrm{DPL}^2_{g,t}(\Pi)$ as being described in Figure 5.1, where the rules $UnitDPL_{\mathcal{L}}$ and $UnitDPL_{\mathcal{R}}$ are being replaced by the rules $Propagate_{\mathcal{L}}$ and $Propagate_{\mathcal{R}}$:

$$\begin{array}{lllll} Propagate_{\mathcal{L}} & (L,\emptyset)_{\mathcal{L}}\|(\Gamma_{\mathcal{L}},\emptyset) & \Rightarrow & (L',\emptyset)_{\mathcal{L}}\|(\Gamma_{\mathcal{L}},\emptyset) & \text{if } l \in \mathcal{P}_{\mathcal{L}}(g(\Pi),L) \\ Propagate_{\mathcal{R}} & (L,R)_{\mathcal{R}}\|(\Gamma_{\mathcal{L}},\Gamma_{\mathcal{R}}) & \Rightarrow & (L,R')_{\mathcal{R}}\|(\Gamma_{\mathcal{L}},\Gamma_{\mathcal{R}}) & \text{if } l \in \mathcal{P}_{\mathcal{R}}(t(\Pi,L),R) \end{array} \tag{6.2}$$

As in the cases of the specific solver, the state $(\emptyset,\emptyset)_{\mathcal{L}}$ is initial. $Propagate_{\mathcal{L}}$ refers to the rules in the generate layer, $Propagate_{\mathcal{R}}$ refers to those in the test layer.

Remark 20. Note that the p-conditions do not change the learned clauses. A p-condition that is applied to the generating (testing) layer does not change the testing (generating) layer.

Remark 21. Again, we see that the transition rules *Backjump*, *Learn*, *Forget* and *Restart* are not considered a propagator condition, as $\mathcal{P}_\mathcal{L}$ and $\mathcal{P}_\mathcal{R}$ are sets of p-conditions like in the single layer graph template.

6.2.1 Approximating and Ensuring Pairs

When using the model in practice, four parameters need to be set, namely the sets $\mathcal{P}_\mathcal{L}$ and $\mathcal{P}_\mathcal{R}$ of p-conditions, the generating function g and the witness function t. To stay in a general form, the authors of [BLM16] introduce approximating and ensuring pairs, that translate into correctness of solvers once properly instantiated.

Definition 55. Let ω and ω_1 be types. A generating function g is ω_1-*approximating* with respect to ω if for any program Π there holds the following:

1. For any stable model L of Π there is a ω_1-model L_1 of $g(\Pi)$ such that $L = L_{1|atoms(\Pi)}$.
2. For any ω_1-model M of $g(\Pi)$, $M_{|atoms(\Pi)}$ is a ω-model of Π.

Example 19. *[BLM16]* The generating function $cnfcomp(\Pi)$ returns a CNF formula that stands for the completion $comp(\Pi)$. The function $cnfcomp$ is *cla*-approximating with respect to *sup*. This holds, because

- any stable model L of Π is also a *cla*-model of $cnfcomp(\Pi)$ and
- any *cla*-model M of $cnfcomp(\Pi)$ is a *sup*-model of Π.

Definition 56. Let ω and ω_1 be types. A witness function t is ω_1-*ensuring* with respect to ω when for any set M of literals covering Π such that $M_{|atoms(\Pi)}$ is a ω-model of Π, $M_{|atoms(\Pi)}$ is a stable model of Π if and only if $t(\Pi, M)$ leads to a program that has no ω_1-model.

Example 20. *[BLM16]* The witness function t^C in Figure 4.2 is *cla*-ensuring with respect to *cla*. Any *sup*-model is a *cla*-model. Therefore, t^C is also *cla*-ensuring with respect to *sup*.

Proposition 16. *Let ω, ω_1 and ω_2 be types, g be a generating function that is ω_1 approximating with respect to ω, t be a witness function that is ω_2-ensuring with respect to ω and Π be a program. The set of all stable models of Π is*

$$\{L_{|atoms(\Pi)} | L \text{ is a } \omega_1\text{-model of } g(\Pi) \text{ and } t(\Pi, L) \text{ has no } \omega_2\text{-models}\}.$$

Definition 57. An *approximating-pair* with respect to ω is a pair $(\mathcal{P}, g)$, which consists of a set of p-conditions and a generating function, where $\mathcal{P}$ is ω_1-enforcing and g is ω_1-approximating with respect to ω.

Example 21. The pair $(\{UnitPropagate\}, cnfcomp(\Pi))$ is an approximating-pair with respect to *sup* as well as to *cla*, because $cnfcomp(\Pi)$ is *cla*-approximating with respect to *sup* and $\{UnitPropagate\}$ is *cla*-enforcing according to Proposition 14.

Definition 58. An *ensuring-pair* with respect to ω is a pair $(\mathcal{P}, t)$, which consists of a set of p-conditions and a witness function, where $\mathcal{P}$ is ω_1-enforcing and t is ω_1-ensuring with respect to ω.

Example 22. The pair $(\{UnitPropagate\}, t^C)$ is an ensuring pair with respect to *cla*, because t^C is *cla*-ensuring with respect to *cla* and $\{UnitPropagate\}$ is *cla*-enforcing according to Proposition 14.

Proposition 17. *[BLM16] Let ω_1 and ω_2 be types. Let g be a generating function, $\mathcal{P}_\}$ be a ω_1-enforcing set of p-conditions, t be a witness function, $\mathcal{P}_\sqcup$ be a ω_2-enforcing set of p-conditions. For a program Π and a state $(l_1 \dots l_{k_1}, r_1 \dots r_{k_2})$ of $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ reachable from the initial state, there hold*

1. *any ω_2-model of $t(\Pi, l_1 \dots l_{k_1})$ satisfies r_i if it satisfies all decision literals $(r_j)^\Delta$ with $j \leq i$ and*
2. *any ω_1-model L of $g(\Pi)$ such that $t(\Pi, L)$ has no ω_2-model satisfies l_i if it satisfies all decision literals l_j^Δ with $j \leq i$.*

Proof. see [BLM16][Lemma 5] □

Proposition 18. *[BLM16] Let ω, ω_1 and ω_2 be types, g be a generating function, that is ω_1-approximating with respect to ω, t be a witness function, that is ω_1-ensuring with respect to ω and Π be a program .Then the set of all stable models of Π is*

$$\{L_{atoms(\Pi)} | L \text{ is a } \omega_1\text{-model of } g(\Pi) \text{ and } t(\Pi, L) \text{ has no } \omega_2\text{-models}\}.$$

Proof. see [BLM16][Propsition 4] □

For the next theorem, we assume that the transition rules $Learn_\mathcal{L}$ and $Learn_\mathcal{R}$ are the ones from Figure 5.2 to ensure acyclicity.

Theorem 6. *Let Π be a program, ω be a type, $(\mathcal{P}_g, g)$ be an approximating-pair with respect to ω and $(\mathcal{P}_t, t)$ be an ensuring-pair with respect to ω. There hold,*

1. *the graph $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ is finite and acyclic,*
2. *any terminal state of $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ reachable from the initial state is either $Failstate$ or of the form $Ok(L)$, with $L_{|atoms(\Pi)}$ being a stable model of Π.*
3. *$Failstate$ is reachable from the initial state if and only if $g(\Pi)$ has no ω-model such that its witness is unsatisfiable.*

Proof. We assume that ω_1 is a type with $\mathcal{P}_g$ being ω_1-enforcing and g being ω_1-approximating with respect to ω and ω_2 is a type with $\mathcal{P}_t$ being ω_2-enforcing and t being ω_2-approximating with respect to ω.

1. In the proof of Theorem 3, we have already seen that the graph $\mathrm{DPL}^2_{g,t}(\Pi)$ with backjumping and learning is finite and acyclic. As the nodes in $\mathrm{DPL}^2_{g,t}(\Pi)$ and $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ are built on the same sets as in the previous theorem that considers backjumping and learning,

namely states relative to $atoms(\Pi)$ and $atoms(t, \Pi, atoms(g(\Pi)))$, and neither L, R, $\Gamma_{\mathcal{L}}$ or $\Gamma_{\mathcal{R}}$ allow repetitions, due to the fact that we use sets, we remain finite.

Now we want to show acyclicity. Graph $\text{DPL}^2_{g,t}(\Pi)$ and $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$, that both include backjumping and learning, share all the transition rules, but $Propagate_{\mathcal{L}}$ and $Propagate_{\mathcal{R}}$ in $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ and $UnitDPL_{\mathcal{L}}$ and $UnitDPL_{\mathcal{R}}$ in $\text{DPL}^2_{g,t}(\Pi)$. For all the transition rules, but the *Propagate* rules, we can follow the lines of the proof of acyclicity in Theorem 3 which includes backjumping and learning. We have seen that the transition rule *Unit* in $\text{DPL}^2_{g,t}(\Pi)$ and *Propagate* with $\mathcal{P} = \{UnitPropagate\}$ coincide. Recall, that to proof acyclicity in CMODELS, we wrote the string L as $L_0 l_1^{\Delta} L_1 \dots l_k^{\Delta} L_k$, where $(l_t^{\Delta})_{1 \leq t \leq k}$ are all the decision literals of L and used the lexicographic order of the sequence $|L_0|, |L_1|, \dots |L_k|$. We only have to look at the remaining propagator conditions, as they are the difference between the earlier proof of acyclicity in Theorem 3 with backjumping and learning. We see, that applying *AllRulesCancelled*, *BackchainTrue* or *Unfounded* only change the length of L_k or R_k, depending on whether *Propagate* is being applied as $Propagate_{\mathcal{L}}$ or $Propagate_{\mathcal{R}}$. Let's assume $L_k(R_k)$ translates to $L'_k(R'_k)$. On applying $Propagate_{\mathcal{L}}$ ($Propagate_{\mathcal{R}}$) with some propagator condition p there holds $|L'_k| = |L_k| + 1$ ($|R'_k| = |R_k| + 1$). Therefor, the other propagator conditions fulfill $v(L, R, s, \Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}}) \leq_{lex} v(L', R', s', \Gamma'_{\mathcal{L}}, \Gamma'_{\mathcal{R}})$ and $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ is acyclic.

2. We have seen in the proof of Theorem 2, that the terminal states of $\text{DPL}^2_{g,t}(\Pi)$ are either *Failstate* or of the form $Ok(L)$, where L is a model of $g(\Pi)$ such that $t(\Pi, L)$ is unsatisfiable. We can use the same argumentation for $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$. For the longer argumentation, see Theorem 2 part 2. Here, I will review the idea: $(L, R)_s \| (\Gamma_{\mathcal{L}}, \Gamma_{\mathcal{R}})$ is not a terminal state, because if $s = \mathcal{L}$ ($s = \mathcal{R}$) a Left-rule or $Cross_{\mathcal{LR}}$ (right-rule, $Conclude_{\mathcal{RL}}$, $Conclude_{\mathcal{R}}$ or $Backjump_{\mathcal{RL}}$) would apply, which would mean that the state is not terminal. Additionally, there is no transition rule that origin at *Failstate* or $Ok(L)$, so those are terminal states. As well as in Theorem 2, the only transition rule that leads to $Ok(L)$ is $Conclude_{\mathcal{R}}$, which is applied if $t(\Pi, L)$ is unsatisfiable. R contains no decision literals and because of Proposition 17(a), $t(\Pi, L)$ has no ω_2-model. Further, because of Proposition 17(b), the consistent set of literals obtained from L is a ω_1-model of $g(\Pi)$. By Proposition 18 $L_{atoms(\Pi)}$ is a stable model of Π.

3. The proof for this part is identical to the third part of proof of Theorem 2. □

We see, that the graph template can be used to define transition systems that result in correct algorithms for deciding whether a program has a stable model or not.

Extending the two-layer graph template to forgetting and restarting is very similar to the previous chapter. Adding the transition rules $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$ from Figure 5.3 to the graph $STTL^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ with backjumping and leraning to create graph $STTLF^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ with backjumping, learning and forgetting. Keep in mind that we lose acyclicity as we can learn and forget the same clause, which leads to a cycle. The same reason would also impair finiteness, which is a feature that we do not want to lose. Therefore, we need a restriction. Theorem 6 changes accordingly:

Theorem 7. *Let* Π *be a program,* ω *be a type,* $(\mathcal{P}_g, g)$ *be an approximating-pair with respect to* ω *and* $(\mathcal{P}_t, t)$ *be an ensuring-pair with respect to* ω*. There hold,*

1. *the graph* $STTLF^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ *is finite if it does not contain an infinite subsequence of only Learn and Forget steps,*
2. *any terminal state of* $STTLF^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ *reachable from the initial state is either* $Failstate$ *or of the form* $Ok(L)$*, with* $L_{|atoms(\Pi)}$ *being a stable model of* Π*.*
3. $Failstate$ *is reachable from the initial state if and only if* $g(\Pi)$ *has no* ω*-model such that its witness is unsatisfiable.*

Proof. Due to the restriction of not having subsequences of inifinite length consisting of only Learn and Forget steps, adding the transition rules $Forget_{\mathcal{L}}$ and $Forget_{\mathcal{R}}$, we can follow the lines of the proof of Theorem 6 to prove finiteness. The additional transition rules do not change the terminal states, because both rules lead to another state, that is known not to be a terminal state. Therefore, we can use the arguments from Theorem 6 for the second and third part of the theorem as well. □

To receive a graph template that captures backjumping, learning, forgetting and restarting, we add the restart rules from Figure 5.4 to the graph $STTLF^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ as additional edges. We call the according graph $STT2^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$. The theorem differs from $STTLF^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ in a sense that we need another restriction to keep finiteness:

Theorem 8. *Let* Π *be a program,* ω *be a type,* $(\mathcal{P}_g, g)$ *be an approximating-pair with respect to* ω *and* $(\mathcal{P}_t, t)$ *be an ensuring-pair with respect to* ω*. There hold,*

1. *the graph* $STT2^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ *is finite if it does not contain an infinite subsequence of only Learn and Forget steps and Restart has increasing periodicity in it,*
2. *any terminal state of* $STT2^{\mathcal{P}_g,g}_{\mathcal{P}_t,t}(\Pi)$ *reachable from the initial state is either* $Failstate$ *or of the form* $Ok(L)$*, with* $L_{|atoms(\Pi)}$ *being a stable model of* Π*.*
3. $Failstate$ *is reachable from the initial state if and only if* $g(\Pi)$ *has no* ω*-model such that its witness is unsatisfiable.*

Proof. The proof follows the lines of Theorem 7. From Proposition 14 we already know that there are not infinitely many occurrences of Restart due to the restriction. Therefore, we remain finite. As the transition rules $Restart_{\mathcal{L}}$ and $Restart_{\mathcal{R}}$ both lead to a state that was showed not to be a terminal state, we can refer to the proof of Theorem 7 to show part 2 and 3 of the theorem. □

6.2.2 The graph template for CMODELS

The instantiation $STTL^{\{UnitPropagate\},t^C}_{\{UnitPropagate\},g^C}$ of the two-layer graph template results in $DPL^2_{g,t}$. They share the same nodes and most transition rules. The difference lies in the form of the transition rule $Propagate_{\mathcal{L}}$ in $STTL^{\{UnitPropagate\},t^C}_{\{UnitPropagate\},g^C}$ and $Unit_{\mathcal{L}}$ in $DPL^2_{g,t}$. But as the set of p-conditions is only $UnitPropagate$, the rule $Propagate_{\mathcal{L}}$ is applicable in

$STTL^{\{UnitPropagate\},t^C}_{\{UnitPropagate\},g^C}$ if and only if $Unit$ is applicable in $DPL^2_{g,t}$. The same statement holds for the case of $Propagate_{\mathcal{R}}$ and $Unit_{\mathcal{R}}$. We see, that the graph template captures CMODELS with backjumping and learning, as well as forgetting and restarting.

In the previous subsection, we have seen, that $g^C = cnfcomp(\Pi)$ is cla-approximating with respect to cla, t^C is cla-ensuring with respect to cla, the pair $(\{UnitPropagate\}, g^C)$ is an approximating-pair with respect to cla and the pair $(\{UnitPropagate\}, t^C)$ is an ensuring-pair with respect to cla.

Instead of the graph $STTL$ with backjumping and learning we can use $STTLF$ with backjumping, learning and forgetting or $STTL2$ with backjumping, learning, forgetting and restarting, accordingly. The previous argumentation does not change. Depending on the used graph, one of the Theorems 6, 7 or 8 holds.

6.2.3 The graph template for GNT

As we have seen in Section 4.3, thy system uses the SMODELS procedure that finds stable models for non-disjunctive logic programs, while CMODELS uses the DPLL procedure to find classical models. Instead of the graph $DPL^2_{g,t}$ for CMODELS, the graph $SML^2_{g(\Pi),t}$ represents GNT. The graph $STTL^{\{UnitPropagate,AllRulesCancelled,BackchainTrue,Unfounded\},t^G}_{\{UnitPropagate,AllRulesCancelled,BackchainTrue,Unfounded\},g^G}(\Pi)$ is the GNT representation using the graph template, where g^G is being described in Equation 4.4 and t^G in Equation 4.5. g^G is sta-approximating with respect to cla and t^G is sta-ensuring with respect to cla according to [BLM16]. Therefore, the pair $(\{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}, g^G)$ is an approximating-pair with respect to cla, because g^D is sta-approximating with respect to cla and $\mathcal{P}_g = \{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}$ is sta-enforcing according to Proposition 14.

The pair $(\{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}, t^G)$ is an ensuring-pair with respect to cla, because t^G is sta-ensuring with respect to cla and $\mathcal{P}_t = \{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}$ is sta-enforcing according to Proposition 14.

Therefore, we can apply Theorem 6 to GNT. When we use $STTLF$ ($STT2$), Theorem 7 (8) applies.

6.2.4 The graph template for DLV

In Section 4.4, we have seen, that DLV is similar to SMODELS without the rule *Unfounded* in the generate layer and DPLL in the test layer, that results in the graph $(SM^* \times DP)_{g(\Pi),t}$, while we still make sure that the transition rules are defined such that disjunctive rules are captured as well.

The graph $STTL^{\{UnitPropagate\},t^D}_{\{UnitPropagate,AllRulesCancelled,BackchainTrue\},g^D}(\Pi)$ represents DLV using the graph template, where g^D is being described in Equation 4.6 and t_G in Equation 4.7.

The generating function g^D is sup-approximating with respect to cla and the witness function t^D is cla-ensuring with respect to cla according to [BLM16].

The pair $(\{UnitPropagate, AllRulesCancelled, BackchainTrue\}, g^D)$ is an approximating - pair with respect to *cla*, because g^D is *sup*-approximating with respect to *cla* and $\mathcal{P}_g = \{UnitPropagate, AllRulesCancelled, BackchainTrue\}$ is *sup*-enforcing according to Proposition 14.

The pair $(\{UnitPropagate\}, t^D)$ is an ensuring-pair with respect to *cla*. We have seen in this subsection, that t^D is *cla*-ensuring with respect to *cla*. The set $\mathcal{P}_t$ is *cla*-enforcing, because of Proposition 14.

We can apply the Theorems 6, 7 and 8 to DLV, depending on what graph we use instead of $STTL$.

6.2.5 A new solver

The graph template can be used to create new abstract solvers, simply by finding new approximating-pairs with respect to a type ω and ensuring-pairs with respect to the same type ω and combining them. Theorem 6 and its extensions Theorem 7 and Theorem 8 provide a proof of correctness and termination. To obtain new solvers we can combine any approximating-pair on the generate layer with any ensuring pair on the test layer with respect to the same type.

Example 23. $STTL^{\{UnitPropagate, AllRulesCancelled, BackchainTrue\}, t^D}_{\{UnitPropagate\}, g^D}(\Pi)$ could be an abstract solver. When we compare it to the graph template for DLV, we see that the propagator conditions for the generate layer and those for the test layer are interchanged. The generate layer is being depicted by the DPLL procedure. The left-rules of the graph are the left-rules of $\mathrm{DPL}^2_{g,t}(\Pi)$ from Section 5.1. The test layer consists of the same right-rules as $SML^2_{g(\Pi),t}$without the transition rule $Unfounded_{\mathcal{R}}$ as stated in Section 5.6.

From Section 6.2.2 we know that the pair $\{\{UnitPropagate\}, g^D\}$ is approximating-pair with respect to *cla*, because we use a generating function g^D, that is shown to be *cla*-approximating in Section 6.2.4. Using Proposition 14 we see that the set $\{UnitPropagate, AllRulesCancelled, BackchainTrue\}$ is *sup*-enforcing and therefore *cla*-enforcing. The test function t^D is *cla*-enforcing according to 6.2.4. Thus, the pair $\{\{UnitPropagate, AllRulesCancelled, BackchainTrue\}, t^D\}$ is an ensuring-pair with respect to *cla* and Theorem 6 holds. Analogously as in the previous sections we can use the graph $STTLF$ ($SST2$) instead of $STTL$, so that Theorem 7 (8) holds.

Example 24. For any pair $(\mathcal{P}, g)$ that is ensuring with respect to *cla* the solver $STTL^{\{UnitPropagate\}, t^C}_{\mathcal{P},g}(\Pi)$ would fulfill Theorem 6. This graph is stated in a more general way as the approximating-pair is not specified directly. It could be any ensuring-pair. The DPLL procedure is used in the test layer. From Section 6.2.2 we know that $\{\{UnitPropagate\}, t^C\}$ is an ensuring-pair with respect to *cla*.

Using $STTLF^{\{UnitPropagate\}, t^C}_{\mathcal{P},g}(\Pi)$ would fulfill Theorem 7 and $STT2^{\{UnitPropagate\}, t^C}_{\mathcal{P},g}(\Pi)$ would fulfill Theorem 8.

7 Conclusion

This thesis has tried to show that we can depict disjunctive answer set solvers with more advanced rules like backjumping, learning, forgetting and restarting in an abstract framework by extending the previous built frameworks for answer set solvers like CMODELS, GNT and DLV to said rules. Additionally, we analysed features like termination and correctness when including these rules. To achieve features like finiteness and acyclicity, we have seen, that sometimes we can or even need to add some restrictions to guarantee those. When we add forgetting and restarting though, we lose acyclicity, because these rules theoretically allow to revisit previously reached states. With the knowledge from the work with the specific solvers, we went into even more abstract dimensions and saw that we can also extend the graph template from [BLM16] that captures these disjunctive answer set solvers and can be used to create new, unknown disjunctive answer set solvers. We have seen that we can guarantee features like termination and completeness for those yet unknown answer set solvers with a single theorem and observed finiteness and acyclicity in the graph template as well, depending on the transition rules that we added previously.

Bibliography

[BLM14] R. Brochenin, Y. Lierler, and M. Maratea. Abstract disjunctive answer set solvers. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, pages 165–170. IOS Press, 2014.

[BLM16] R. Brochenin, Y. Lierler, and M. Maratea. Abstract disjunctive answer set solvers via templates. In *Theory and Practice of Logic Programming*, volume 16, pages 465–497. Cambridge University Press, 2016.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications*. North-Holland, 1976.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5, pages 394–397. Association for Computing Machinery, 1962.

[DM17] C. Dodaro and M. Maratea. Nurse scheduling via answer set programming. In M. Balduccini and T. Janhunen, editors, *Logic programming and nonmonotonic reasoning*, volume 10377. Springer, 2017.

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning*, volume 1348, pages 169–181. Springer, 1997.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of ACM*, volume 7, pages 201–215. Association for Computing Machinery, 1960.

[EAP12] E. Erdem, E. Aker, and V. Patoglu. Answer set programming for collaborative housekeeping robotics: Representation, reasoning and execution. In *Intelligent Service Robotics*, volume 5, pages 275–291. Springer, 2012.

[FFea18] A. Falkner, G. Friedrich, and Schekotihin et al. Industrial applications of answer set programming. In *Künstliche Intelligenz*, volume 32, pages 165–176. Springer, 2018.

[GKSS08] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In F. Van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of International Logic Programming Converence and Symposium*, pages 1070–1080. MIT Press, 1988.

[GLM06] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. In *Journal of Automated Reasoning*, volume 36, pages 345–377. Springer, 2006.

[GLM08] E. Giunchiglia, N. Leone, and M. Maratea. On the relation among answer set solvers. In *Annals of Mathematics and Articial Intelligence*, volume 53, pages 169–204. Kluwer Academic Publishers, 2008.

[GM05] E. Giunchiglia and M. Maratea. On the relation between answer set and sat procedures (or, between smodels and cmodels). In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming*, volume 3668, pages 37–51. Springer, 2005.

[GS06] M. Gebser and T. Schnaub. Tableau calculi for answer set programming. In S. Etalle and M. Truszczynski, editors, *Proceedings of the 22nd International Conference on Logic Programming*, volume 4079, pages 11–25. Springer, 2006.

[GS13] M. Gebser and T. Schnaub. Tableau calculi for logic programs under answer set semantics. In *ACM Transactions on Computational Logic*, volume 14, pages 15:1–15:40. Association for Computing Machinery, 2013.

[Jan] T. Janhunen. gnt (generate'n'test): A solver for disjunctive logic programs. http://www.tcs.hut.fi/Software/gnt/. Accessed: 2019-03-06.

[JNS+06] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. In *ACM Transactions on Computunational Logic*, volume 7, pages 1–37. Association for Computing Machinery, 2006.

[LFP+06] N. Leone, W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. In *ACM Transactions on Computational Logic*, volume 7, pages 499–562. Association for Computing Machinery, 2006.

[Lie05] Y. Lierler. cmodels - sat-based disjunctive answer set solver. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 3662, pages 447 – 451. Springer, 2005.

[Lie10] Y. Lierler. *SAT-based Answer Set Programming*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2010.

[Lie11] Y. Lierler. Abstract answer set solvers with backjumping and learning. In *Theory and Practice of Logic Programming*, volume 11, pages 135–169. Cambridge University Press, 2011.

[Lie17] Y. Lierler. Basics behind answer sets. http://works.bepress.com/yuliya_lierler/71/, 2017. Accessed: 2020-02-09.

[Lif99] V. Lifschitz. Answer set planning. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 1730, pages 23–37. Springer, 1999.

[Lif08] V. Lifschitz. What is answer set programming?. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 3, pages 1594–1597. AAAI Press, 2008.

[LRS97] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. In *Information and Computation*, volume 135, pages 69–112. Elsevier, 1997.

[LT11] Y. Lierler and M. Truszczynski. Transition systems for model generators - a unifying approach. In *Theory and Practice of Logic Programming*, volume 11, pages 629–646. Cambridge University Press, 2011.

[LZ02] F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. In *Eighteenth National Conference on Artificial Intelligence*, volume 157, page 112–117. American Association for Artificial Intelligence, 2002.

[MT99] V.W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In Apt K.R., Marek V.W., Truszczynski M., and Warren D.S., editors, *The Logic Programming Paradigm. Artificial Intelligence*, pages 375–398. Springer, 1999.

[NBG+01] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In Ramakrishnan I.V., editor, *Practical Aspects of Declarative Languages*, pages 169–183. Springer, 2001.

[Nie99] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Annals of Mathematics and Artificial Intelligence*, volume 25, pages 241–273. Kluwer Academic Publishers, 1999.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). In *Journal of the ACM*, volume 53, pages 937–977. Association for Computing Machinery, 2006.

[RFL06] F. Ricca, W. Faber, and N. Leone. A backjumping technique for disjunctive logic programming. In *AI Communications*, volume 19, pages 155–172. IOS Press, 2006.

[Tep17] E. Teppan. On the complexity of the partner units decision problem. In *Artificial Intelligence*, volume 248, pages 112 – 122. Elsevier, 2017.