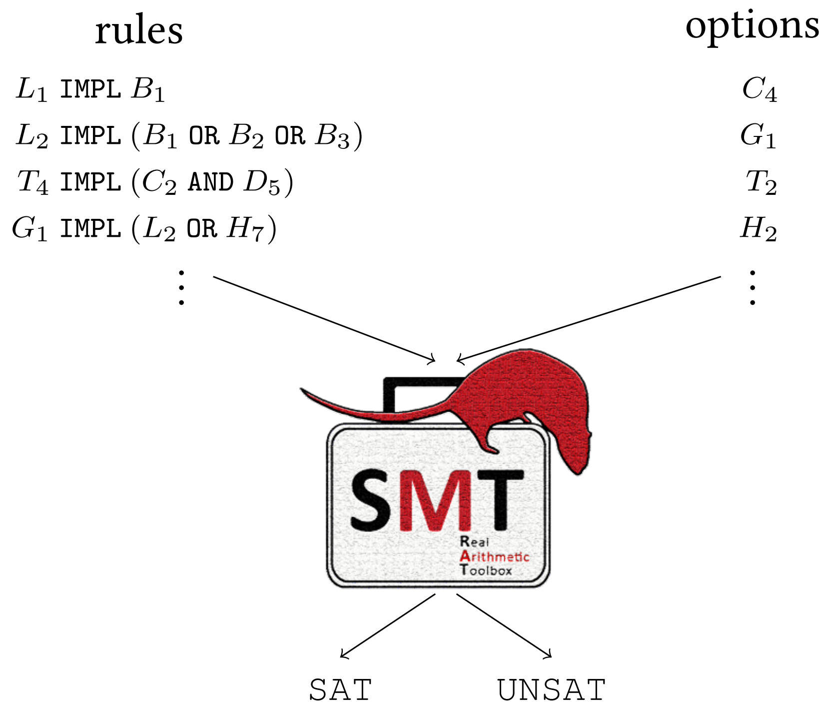
Master of Science Thesis

Using SAT Solvers for Industrial Combinatorial Problems

Matthias Volk

November 26, 2015

LuFG Theory of Hybrid Systems
RWTH Aachen University



Supervisors:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Advisor:
Dipl. Inform. Florian Corzilius

Abstract

SAT solving is the task of checking whether a given propositional logic formula has a solution, i. e., if there exists an assignment to all Boolean variables s. t. the whole formula is satisfied. As a lot of problems can be easily encoded into propositional logic, SAT solving is a general purpose approach which can be used in many different areas.

In this thesis we consider the application of SAT technologies to the area of product configurations in a manufacturing company. The main challenge here is the diversity of the manufactured products. Each product has different features which each have several options. By choosing options we can construct a custom product. However, it must be ensured that this product can be built. This is handled by checking whether a certain set of rules constraining the product and its options is satisfied. As a SAT solver is predestined for solving this task we used our own solver `SMT-RAT` for improving the performance of the product validity checks in the manufacturing company.

Alongside the order checking, i. e., if a given product can be built, two additional tasks were tackled. First the realizability checking searches for those options which can never be chosen and therefore indicate inconsistencies in the set of internal rules. We solve this task by generating lemmas inside our solver which give the reason for certain variable assignments and therefore state those variables which cannot be satisfied.

Last the variance generation handles the development of new components. For a set of combinations of options we have to identify those combinations which are realizable according to the internal rules and those which are not possible. Then for the realizable combinations new components need to be developed. We handled this task by solving the All-SAT problem where we search not for only one but all possible solutions to a problem.

Acknowledgements

I would like to especially thank Christian J. who was our main contact on the part of the company. It was a great experience and joy collaborating on this project and working together on integrating our implementations. Furthermore a great thank to you and your wife for the great hospitality I received during my visits at the company.

I also would like to thank my Advisor Florian Corzilius for the help with my thesis and the implementation, Gereon Kremer for insights into `SMT-RAT` and especially helping me to find bugs in there via delta debugging, Prof. Erika Ábrahám for enabling this cooperation, and for the support of her and the whole THS group during this project.

Eidesstattliche Versicherung

Matthias Volk

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel "*Using SAT Solvers for Industrial Combinatorial Problems*" selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Matthias Volk
Aachen, den 26. November 2015

Belehrung:
§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Matthias Volk
Aachen, den 26. November 2015

Contents

1 Introduction **1**
1.1 History of satisfiability solving . . . 1
1.2 Special problem cases . . . 1

2 Propositional logic **3**
2.1 Syntax and semantics . . . 3
2.2 Conjunctive normal form . . . 4
2.3 Tseitin's encoding . . . 4

3 Problem statement **7**
3.1 Application field: A manufacturing company . . . 7
3.2 SAT encoding . . . 7
3.3 Prospect: Satisfiability modulo theories . . . 9
3.4 Objectives . . . 10
3.4.1 Order checking . . . 10
3.4.2 Realizability checking . . . 11
3.4.3 `VarGen` checking . . . 12

4 SAT solving **13**
4.1 Enumeration . . . 13
4.2 Resolution . . . 13
4.3 DPLL . . . 14
4.3.1 Boolean constraint propagation . . . 16
4.3.2 Enumeration . . . 16
4.3.3 Conflict analysis . . . 17
4.3.4 Complete example . . . 18

5 Incremental SAT solving **21**
5.1 Interfaces . . . 21
5.2 Order checking . . . 22

6 Lemma generation **23**
6.1 Naive algorithm . . . 23
6.2 Improved algorithm . . . 23
6.3 Final algorithm . . . 24
6.3.1 Complete example . . . 28

7 All-SAT **31**
7.1 Naive algorithm . . . 31
7.2 All-SAT . . . 32
7.2.1 Final algorithm . . . 32
7.2.2 Complete example . . . 32

8 Implementation **35**
8.1 Architecture . . . 35
8.2 SMT-RAT . . . 35

9 Evaluation **37**
9.1 Order checking . . . 37
9.2 `PCC` checking . . . 38
9.3 `VarGen` checking . . . 38

10 Conclusion and future work **41**

10.1 Conclusion 41

10.2 Future work 41

Bibliography **43**

1 Introduction

In this thesis we will apply SAT solving techniques to product configuration problems in a manufacturing company. Due to protecting business secrets, the name of the company we worked with along with specific details will not be disclosed.

1.1 History of satisfiability solving

The *satisfiability (SAT)* problem is the task to decide whether there exists a satisfying solution to a propositional logic formula. A propositional logic formula is built from Boolean variables and the common logical operations negation, conjunction and disjunction. A solution for such a propositional logic formula assigns every Boolean variable a truth value `true` or `false`, such that the whole formula evaluates to `true`. If such a solution can be found, the formula is satisfiable, otherwise it is unsatisfiable.

There are two main methods of solving the SAT problem: enumeration and resolution. The first method enumerates all possible variable assignments and checks each of them for satisfiability. This works well for satisfiable formulas, but in case of unsatisfiability all possible assignments have to be checked, leading to an exponential time consumption.

The second approach tries to find a proof for unsatisfiability by applying the resolution inference rule. This could take exponentially many possible resolution steps as well, whereas possibly a subset of them could already prove unsatisfiability.

Work on SAT solvers started with the introduction of the *Davis-Putnam algorithm* [DP60] in 1960, which uses resolution. It was refined in 1962 with the famous *Davis-Putnam-Loveland-Logemann (DPLL) framework* [DLL62], combining enumeration, propagation and resolution. Afterwards, resolution was further improved by Robinson [Rob65] in 1965.

Later another famous theoretical breakthrough showed the NP-completeness of the SAT problem and was done independently by Cook [Coo71] and Levin [Lev73].

An alternative approach to DPLL are stochastic solvers which were led for many years by `GSAT` and `WALKSAT` [SLM92]. These solvers assign random values to variables and then check for satisfiability. If the formula is not yet satisfied the solver flips a variable and checks again.

Another approach is the use of *Binary Decision Diagrams (BDD)*, a canonical graph-based date structure representing Boolean formulas, which were introduced in 1959 [Lee59] and further developed by Akers [Ake78] and Bryant [Bry86].

During the last decades a lot of progress has happend in the research area of SAT solving with extensions to *SAT Modulo theories (SMT)* solving, where propositional logic is extended to theories, e. g., linear and non-linear real arithmetics. Furthermore, the improved performance of SAT and SMT solvers led to an easier application of solving technologies in the industry.

In the last 20 years a lot of different solvers emerged, beginning with the solver `GRASP` [MSS96] which was the fastest SAT solver from 1996 to 2000. Using the knowledge of previous solvers and improving upon the original DPLL framework other solvers such as `Chaff` [Mos+01], `BerkMin` [GN07] and `MiniSat` [ES03] were developed.

A good indicator of the current status in SAT and SMT solving is the annual *SAT competition* [Bel+14] with SMT solvers such as `Yices` [Dut14], `CVC4` [Bar+11], `MathSAT` [Cim+13], `Z3` [DMB08] and our own solver `SMT-RAT` [Cor+15b].

1.2 Special problem cases

SAT and SMT solving are applied in a lot of fields: from bounded model checking [Bie+99] and model checking of parametric probabilistic systems [Deh+15] over automated termination analysis [Fuh+07] and planning [KS92] to package management in software distributions [Tuc+07].

In the industry especially the problem of *product configurations* is important where it must be ensured that a product specified by certain configuration options is producible. Here a set of rules is given constraining the different product types and their options. When choosing several options for a product, the task is now to determine, whether such a configuration is possible, i. e., if it is satisfiable for the given rules.

Previous work in this area already used SAT solvers [SKK00] [SKK01], BDDs [MA04] or methodologies from artificial intelligence [Fal+11] to accomplish this task. In our setting we will use SAT solvers as well, mainly our own solver `SMT-RAT`. This gives us the opportunity to implement custom algorithm in the solver tailored to our specific tasks.

Beside the product configuration we also have to check the *realizability of all options*. For every possible option we have to ensure that it can be chosen in at least one realizable configuration. Otherwise this option is unrealizable and indicates inconsistencies in the set of rules. We solve this task by generating lemmas in the solver, stating the reason for certain variable assignments.

Finally we have the task of *variance generation (`VarGen`)*. Here for a given set of options we have to determine those combinations of options which lead to realizable configurations, i. e., whose propositional encoding along with the corresponding rules is satisfiable. This problem is also known as All-SAT, where we do not search for only one solution to a SAT problem but for all solutions. Work on All-SAT has already be done in [GSY04] or [Yu+14].

Using an SMT solver instead of a plain SAT solver also gives us the opportunity to extend our used logic later on to a more expressive one. A prospect on this will be stated in Section 3.3 on page 9.

The outline of the thesis is as follows. We start by giving a short introduction into Boolean logic in Chapter 2. In Chapter 3 we introduce the three tasks given to us in detail. Before explaining our approaches for the three problems we give an introduction into SAT solving, especially the DPLL framework, in Chapter 4. Chapter 5 explains the benefit of incremental SAT solving for the product configuration problem. In Chapter 6 we describe the generation of lemmas to help solving the realizability problem. The `VarGen` task is solved in Chapter 7 with use of All-SAT. In Chapter 8 we discuss our implementation and its integration within the company. The evaluation of our algorithms is described in Chapter 9. We conclude in Chapter 10 and give an outlook into possible future work.

2 Propositional logic

2.1 Syntax and semantics

Propositional logic formulas are constructed from Boolean variables with standard operations as *conjunction*, *disjunction* or *negation*. The formal syntax of propositional logic formulas is as follows.

Definition 2.1 (Propositional logic)**.** The syntax of formulas in *propositional logic* is defined by the following rules:

$atom : Boolean\ identifier$	Boolean variable
$\mid \texttt{true} \mid \texttt{false}$	Boolean constants
$formula : atom$	atom
$\mid (\neg formula)$	negation
$\mid (formula \wedge formula)$	conjunction
$\mid (formula \vee formula)$	disjunction

▲

Other "syntactic sugar" can be expressed by using these operators. For example we can write the operator "*implies*" as

$$(a \Rightarrow b) := ((\neg a) \vee b)$$

The operator "*if and only if (iff)*" can be written as

$$(a \Leftrightarrow b) := ((a \Rightarrow b) \wedge (b \Rightarrow a))$$

For better readability we often omit brackets and use the inherent binding of operators: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ have decreasing binding strength, i. e., $\neg$ binds stronger than $\wedge$, etc.

We introduce some further notions of variable assignments and satisfiability. For a propositional logic formula φ let $Var(\varphi)$ denote the set of Boolean variables appearing in φ. Let $\mathbb{B}$ denote the Boolean domain $\{\texttt{false}, \texttt{true}\}$ which we also see alternatively as binary domain $\{0, 1\}$.

Definition 2.2 (Assignment)**.** For a propositional logic formula φ an *assignment* is a partial function $Var(\varphi) \rightharpoonup \mathbb{B}$ mapping the variables of φ to elements of $\mathbb{B}$.

An assignment to φ is *full* if all variables of φ are assigned and *partial* otherwise. ▲

Definition 2.3 (Satisfiability, validity, contradiction)**.** A formula φ is *satisfiable* if there exists a full assignment for it that evaluates φ to $\texttt{true}$.

A formula φ is a *contradiction* if it is not satisfiable.

A formula φ is *valid*, i. e., a *tautology*, if it evaluates to $\texttt{true}$ under all full assignments. ▲

Lemma 2.1 (Validity)**.** *A formula φ is valid if and only if $\neg\varphi$ is a contradiction.*

Some important rules for transforming propositional logic formulas are *De Morgan's rules* and the *distributive laws*.

Definition 2.4 (De Morgan's rules)**.**

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

where $\equiv$ denotes equivalence w. r. t. the set of satisfying assignments. ▲

Definition 2.5 (Distributive laws)**.**

$$a \land (b \lor c) \equiv (a \land b) \lor (a \land c)$$
$$a \lor (b \land c) \equiv (a \lor b) \land (a \lor c)$$
▲

Using these rules we can see that the operator $\lor$ is not necessarily needed in Definition 2.1 as it can be expressed as $a \lor b \equiv \neg(\neg a \land \neg b)$.

2.2 Conjunctive normal form

To better handle propositional logic, several *normal forms* exist introducing a standard way of expressing propositional logic formulas. The base case in all these normal forms are *literals*.

Definition 2.6 (Literal, clause)**.** A *literal* is either an atom a or its negation $\neg a$. A literal is *negative* if it is a negated atom and *positive* otherwise.
A *clause* is a disjunction of literals, i. e., has the form

$$\bigvee_k l_k$$
▲

The normal form which is usually used as input to SAT solvers is the *conjunctive normal form* (CNF).

Definition 2.7 (CNF)**.** A formula is in *conjunctive normal form* if it is a conjunction of clauses, i. e., it is of the form

$$\bigwedge_i (\bigvee_j l_{ij})$$

where l_{ij} is the j-th literal in the i-th clause. ▲

We now can use this form in our tool. For example checking satisfiability of a formula in CNF reduces to checking that at least one literal in every clause is satisfied.

Theorem 2.2 (Transformation into CNF)**.** *Every propositional logic formula can be transformed into an equivalent formula in CNF.*

Proof. The transformation follows several steps:

1. Transform all syntactic sugar like $\Rightarrow$ into their corresponding formulas expressed by $\land$, $\lor$, $\neg$.
2. Repeatedly apply De Morgan's rules (see Definition 2.4) to move all negation inwards.
3. Use the distributive laws (see Definition 2.5) to move $\lor$ inwards over $\land$. □

However, this approach comes with a main disadvantage: Because of the application of the distributive laws the size of the formula can grow exponentially during the transformation.

Example 2.1 (Exponentially growth during distributive laws)**.** Consider the following formula with n clauses:

$$(a_1 \land b_1) \lor (a_2 \land b_2) \lor \cdots \lor (a_n \land b_n)$$

Applying the distributive laws leads to a formula with 2^n clauses:

$$(a_1 \lor a_2 \lor \cdots \lor a_n) \land (b_1 \lor a_2 \lor \cdots \lor a_n) \land (a_1 \lor b_1 \lor \cdots \lor a_n) \land \cdots \land (b_1 \lor b_2 \lor \cdots \lor a_n) \land (b_1 \lor b_2 \lor \cdots \lor b_n)$$ ♦

2.3 Tseitin's encoding

To avoid the exponential growth in the CNF transformation we use *Tseitin's encoding* [Tse68] which bounds the growth of the formula linearly. However, this transformation also introduces n new

variables, where n is the number of logical gates in the formula. The value of every newly introduced variable is constrained to be equal to the represented gate. This is best illustrated by an example.

Example 2.2 (Tseitin's encoding)**.** Consider the formula

$$a \Rightarrow (b \wedge \neg c)$$

We introduce a new variable for every logical gate:

$$t_1 \text{ for } \Rightarrow,\ t_2 \text{ for } \wedge \text{ and } t_3 \text{ for } \neg$$

This is illustrated in the *syntax derivation tree* in Figure 2.1.

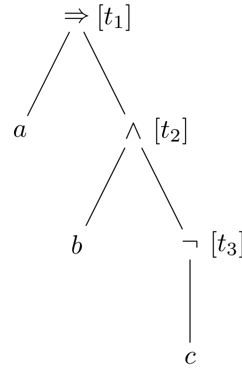


Figure 2.1: Derivation tree in Tseitin's encoding.

Next we constrain the value of these variables according to their gate:

$$t_1 \Leftrightarrow (a \Rightarrow t_2) \tag{2.1}$$
$$t_2 \Leftrightarrow (b \wedge t_3) \tag{2.2}$$
$$t_3 \Leftrightarrow (\neg c) \tag{2.3}$$

Equation (2.1) can be rewritten in CNF as

$$\begin{aligned} & t_1 \Leftrightarrow (a \Rightarrow t_2) \\ \equiv & (\neg t_1 \vee \neg a \vee t_2) \wedge (a \vee t_1) \wedge (t_1 \vee \neg t_2) \end{aligned}$$

Rewriting the other two formulas in CNF and asserting the topmost operator t_1 we get the final CNF formula with 9 clauses and 3 new variables:

$$\begin{aligned} & t_1 \wedge (t_1 \Leftrightarrow (a \Rightarrow t_2)) \wedge (t_2 \Leftrightarrow (b \wedge t_3)) \wedge (t_3 \Leftrightarrow (\neg c)) \\ \equiv\ & t_1 \wedge (\neg t_1 \vee \neg a \vee t_2) \wedge (a \vee t_1) \wedge (t_1 \vee \neg t_2) \wedge (\neg t_2 \vee b) \wedge (\neg t_2 \vee t_3) \wedge (\neg b \vee \neg t_3 \vee t_2) \\ & \wedge (\neg t_3 \vee \neg c) \wedge (c \vee t_3) \end{aligned}$$ ♦

Using Tseitin's encoding a propositional logic formula is transformed into an *equisatisfiable* CNF.

Definition 2.8 (Equisatisfiability)**.** Two formulas φ_1 and φ_2 are *equisatisfiable* iff

$$\varphi_1 \text{ is satisfiable } \Leftrightarrow \varphi_2 \text{ is satisfiable}$$ ▲

Thus, Tseitin's encoding does not change the satisfiability of the formula while only having a linear growth in the size of the formula. From now on we assume that every given propositional logic formula is in CNF.

3 Problem statement

In this chapter we introduce the task and corresponding problem statements given to us from the manufacturing company.

3.1 Application field: A manufacturing company

For capital goods the main challenge is the fact that every product is unique. In contrast to consumer goods where there exists a set of default product configurations on which some features can be added or modified, in our context every product is constructed on its own. This offers greater flexibility in constructing the best product for a given requirement but also leads to the challenge of determining whether chosen components fit together and make a realizable product. Thus, our main task is to check whether a given product configuration is realizable.

3.2 SAT encoding

To formalize the realizability of a product configuration a set of rules is introduced which formalize the constraints on the interaction of the different components.

For example consider batteries and battery holders. If the battery holder has a certain length it might only fit one battery or it might fit two batteries consecutively but only if the batteries are not too long. This can be expressed in a rule.

Example 3.1 (Rules for batteries)**.** For the different battery configurations we introduce three options B_1, B_2 and B_3. Let B_1 indicate that we have one small battery, B_2 indicates one long battery and B_3 indicates two small batteries in series.

Additionally we have two options concerning the length of the battery holder L_1 and L_2 where L_1 indicates a small battery holder and L_2 a larger battery holder.

Using these sets of variables we now can establish some rules for constraining the batteries to the space of the battery holder.

For example the small battery holder can only have one small battery:

$$L_1 \texttt{ IMPL } B_1$$

where `A IMPL B` represents the implication $A \Rightarrow B$.

The slightly larger battery holder can have up to one large battery or two small batteries:

$$L_2 \texttt{ IMPL } (B_1 \texttt{ OR } B_2 \texttt{ OR } B_3)$$ ♦

In the existing solution the rules constitute the so called `PCC` (*Product Configuration Constraints*) for each *product type*. Here for every available *feature*, e. g., batteries, the different *options* for this feature are listed, e. g., one small battery, two large batteries, etc. These indicate an *exactly-one* relationship, i. e., for every feature exactly one of the options has to be taken.

Example 3.2 (Encoding features and options)**.** Assume we have the feature "battery" from Example 3.1 encoded as F_1 with its three options B_1, B_2 and B_3. Then the intuitive encoding of "exactly one" ($\exists!$) would look as follows.

$$\exists! Opt \in F_1.\ Opt \wedge \forall Opt' \in F_1 \setminus \{Opt\}.\ \neg Opt'$$

However, we cannot express this directly in propositional logic. Therefore we split this constraint into two different constraints which both have to be satisfied simultaneously.

1. "At least one":

$$B_1 \lor B_2 \lor B_3$$

2. "Not more than one":

$$\begin{aligned}&\neg(B_1 \land B_2) \land \neg(B_1 \land B_3) \land \neg(B_2 \land B_3)\\ \equiv&(\neg B_1 \lor \neg B_2) \land (\neg B_1 \lor \neg B_3) \land (\neg B_2 \lor \neg B_3)\end{aligned}$$

♦

In general "exactly one" over a set of options $O_1, \ldots, O_n$ can be encoded as:

$$(\bigvee_{1 \le i \le n} O_i) \land (\bigwedge_{\substack{1 \le i < n \\ i < k \le n}} (\neg O_i \lor \neg O_k))$$

Besides encoding options for features there are also additional rules capturing the relationships between different options, e. g., options excluding each other.

Example 3.3 (Rules for feature)**.** Consider the following `PCC` rule.

$$P_1 \quad F_1 \quad Seq_2 \quad O_2 \quad W_{01}^{15}$$

The first entry P_1 indicates the product type which is also the name of the current `PCC`, the second entry F_1 indicates the feature and the third entry Seq_2 stands for the so called sequence number. The fourth entry specifies the option O_2 and the fifth entry W_{01}^{15} encodes the first week of the year 2015, the week from which on this option can be chosen.

All other options for this feature are encoded similarly. The main difference is the different option in the fourth entry. ♦

Further on the `PCC` includes the rules characterizing the interaction between different features and/or options. These rules can be appended to the existing option as:

$$P_1 \quad F_1 \quad Seq_2 \quad O_2 \quad W_{01}^{15} \quad \text{IMPL} \quad O_3$$

This indicates that choosing option O_2 also implies selecting option O_3. Using `EXCL` we can exclude options:

$$P_1 \quad F_1 \quad Seq_2 \quad O_2 \quad W_{01}^{15} \quad \text{EXCL} \quad O_5$$

where `A EXCL B` stands for the exclusion $A \Rightarrow \neg B$. As choosing option O_2 excludes option O_5 this option O_5 cannot be selected anymore.

All options after an `IMPL` or `EXCL` are combined with `AND`.

Example 3.4 (Rules AND)**.** Consider the rule

$$P_1 \quad F_1 \quad Seq_2 \quad O_2 \quad W_{01}^{15} \quad \text{IMPL} \quad O_3, O_4$$

This translates to

$$O_2 \Rightarrow O_3 \land O_4$$

♦

There can be rules with the same option on the left-hand side and the same sequence number. This indicates that all rows corresponding to this option should be combined with `OR`.

Example 3.5 (Rules: OR)**.** Consider the following rules:

$$\begin{array}{lllllll}P_1 & F_1 & Seq_2 & O_2 & W_{01}^{15} & \text{IMPL} & O_3, O_4\\ P_1 & F_1 & Seq_2 & O_2 & W_{01}^{15} & \text{IMPL} & O_6, O_7\end{array}$$

This translates to

$$O_2 \Rightarrow (O_3 \land O_4) \lor (O_6 \land O_7)$$

♦

If the rules have the same option on the left-hand side but different sequence numbers, the rules do not hold at the same time. Here we consider the week in the fifth entry which indicates the beginning week of a rule. The start week of a higher sequence number is the end week of a lower sequence number, i. e., the rule of the lower sequence number is not valid anymore.

Example 3.6 (Rules: Effectivity date)**.** Consider the following rules:

$$\begin{array}{lllllll} P_1 & F_1 & Seq_2 & O_2 & W_{01}^{15} & \text{IMPL} & O_3, O_4 \\ P_1 & F_1 & Seq_3 & O_2 & W_{21}^{15} & \text{IMPL} & O_6, O_7 \end{array}$$

This translates to the following rules with effectivity intervals:

$$O_2 \Rightarrow (O_3 \wedge O_4) \text{ is valid in } [01/15, 21/15)$$
$$O_2 \Rightarrow (O_6 \wedge O_7) \text{ is valid in } [21/15, \infty)$$

♦

These effectivity intervals cannot directly be encoded in propositional logic. Thus, for every effectivity interval $[W_l^k, W_n^m)$ we introduce a new Boolean variable $W_{n/m}^{l/k}$ which is true, if the current week is in the encoded interval.

Example 3.7 (Encoding of effectivity dates)**.** Consider the rules from Example 3.6. When encoding these rules we introduce two new Boolean variables $W_{21/15}^{01/15}$ and $W_{\infty}^{21/15}$. Using these we can encode the rules as follows.

$$W_{21/15}^{01/15} \Rightarrow (O_2 \Rightarrow (O_3 \wedge O_4))$$
$$W_{\infty}^{21/15} \Rightarrow (O_2 \Rightarrow (O_6 \wedge O_7))$$

In the preprocessing we then would set these variables to capture the current week. If the current week is $13/15$ we would make the following assignment beforehand:

$$\{W_{21/15}^{01/15}, \neg W_{\infty}^{21/15}\}$$

Therefore the second rule is already satisfied and does not influence the SAT solving, i. e., it behaves as if not active. ♦

The current implementation consists of several 100 different `PCCs` which each have approximately 700-1500 rules. All in all we have to deal with 210,000 rules in our context.

As one can see the rules in the `PCC` are already similar to propositional logic formulas. Nevertheless the existing algorithm for determining the realizability of a product configuration does not use the advantages of state-of-the-art SAT solving but is a custom algorithm. Therefore our first important task was to transform these rules into propositional logic and then to solve the problems using SAT solving. This will allow to have a better maintainable set of rules and the availability of highly tested and very efficient SAT solving algorithms to choose from for tackling the three main problems as explained in Section 3.4.

3.3 Prospect: Satisfiability modulo theories

Before stating the main problems we will give a short outlook on a possible future extension of the current rules into a more expressive logic such as *satisfiability modulo theories (SMT)*.

Satisfiability modulo theories (SMT) is an extension on propositional logic where some of the Boolean variables are replaced by predicates of a certain theory, e. g., equality logic or linear arithmetic.

An SMT solver uses a SAT solver as core decision engine [NOT05]. First every predicate of a theory is replaced by a new Boolean variable. As the formula is now in propositional logic we can solve it with a SAT solver. If the formula is satisfiable we get an assignment for each variable. We invoke a so called *theory solver* to check the consistency of the set of predicates whose corresponding Boolean variables are assigned to `true` and, in case the input formula contains also negated (in)equalities, the negations of those constraints, whose Boolean variables are assigned to `false`.

In our context the theory of *linear real arithmetic (LRA)* would be most helpful. LRA covers linear equalities and inequalities over the real numbers.

Example 3.8 (LRA)**.** An example LRA formula is:

$$((2x_1 + 3x_2 = 4x_3) \vee (x_2 - 4x_3 \geq 0)) \wedge \neg(3x_1 + 2x_2 < x_3) \qquad \blacklozenge$$

LRA is more expressive than propositional logic but its conjunctive quantifier-free fragment can still be solved efficiently. The common algorithm for checking a set of linear equalities and inequalities is based on the first phase of the *Simplex* method [Dan63] which can be adapted to work in an SMT solver [DDM06a][DDM06b].

Using LRA can be more intuitive for human interactions than formulas in propositional logic, uses much less variables and the encoding of the problems described in Section 3.4 might be solved more efficiently than the propositional encoding.

Our previous Example 3.1 on page 7 could be expressed as an SMT formula with the use of LRA.

Example 3.9 (Rules for batteries in SMT)**.** Now the battery holder spaces are expressed as constants l_1 and l_2 and the length of every battery is expressed as constants b_1 and b_2 with b_1 being the length of a small battery and b_2 the length of a long battery. We use the real-valued variables n_{b_i} counting the number of batteries of type b_i.

Then our rules would be the following linear inequalities which have to be satisfied:

$$(l = l_1 \vee l = l_2) \wedge (n_{b_1} \cdot b_1 + n_{b_2} \cdot b_2 \leq l)$$

The left-hand-side of each inequality determines the required length for the chosen battery holder. Comparing it with the actual length of the battery holders we can determine whether the configuration is producible. In general the sum b of the length of batteries must not be greater than the length l of the battery holder:

$$(\bigvee_i l = l_i) \wedge (\sum_j n_{b_j} \cdot b_j \leq l) \qquad \blacklozenge$$

We see that we need fewer variables and especially fewer rules in our example when using SMT instead of SAT. In case of SAT we have to check all combinations of options and their possible dependencies which then might have to be captured in the rules. This has to be done on a fairly technical level and therefore is fault-prone. In SMT we can formulate the dependencies on a more intuitive level and only need one rule. Then the dependencies between the options are implicitly encoded in the SMT rule and do not have to be stated explicitly anymore. This leads to greater maintainability. If a battery length changes, only the value of one constant has to be changed whereas with the previous SAT encoding every rule considering batteries has to be rewritten by an expert.

3.4 Objectives

We now explain our three main tasks given to us from the company: order checking, realizability checking and `VarGen`.

3.4.1 Order checking

The first problem is the previously mentioned problem to determine whether a certain product configuration is producible, i. e., consistent with a given set of rules. Here we get the `PCC` of the product type, a list of chosen options and the delivery date as input. The output is the answer if the chosen configuration is realizable.

As seen before we can transform the whole set of rules given in the `PCC` into propositional logic. Using these rules and asserting all the given options we can solve the problem with a state-of-the-art SAT solver as it is only requires satisfiability checking.

Example 3.10 (Order checking)**.** In the following, for better understanding we will not list the concrete `PCC` rules but a part of the already transformed propositional logic formula without the

effectivity date and the feature encoding. Then the input PCC looks as follows:

$$O_1 \Rightarrow O_2 \land O_3$$
$$O_5 \Rightarrow \neg O_2$$
$$O_4 \Rightarrow O_1$$

The list of chosen options is O_3, O_4, O_5.

We find out that this order is unrealizable. Choosing option O_4 leads to choosing option O_1 which implies taking option O_2. But as option O_5 forbids taking option O_2 we have reached a contradiction. ♦

Details on this process can be found in Chapter 5 on page 21.

Solving this problem on its own only takes $10 - 500$ milliseconds but the main challenge here is bulk checking. Due to rule modifications, every weekend all orders in the foreseeable future have to be checked which can lead to 50,000 checks. At the moment this bulk check takes up to 6 hours, but as the PCCs become more complex in the future this time consumption might increase drastically. Furthermore if an error occurs during these checks, some corrections on the PCCs might be necessary and all checks will have to be done all over again. Therefore the checking of one order has to be as fast as possible as otherwise this could amount to a huge delay when checking all orders.

3.4.2 Realizability checking

The second problem is the realizability check for PCCs. The PCCs are constantly changing due to input from the engineers as features and options change. Therefore it is necessary to check the validity of the given rules to avoid contradictions. In particular every option for each feature must be selectable in at least one consistent configuration, i. e., there should be no option which cannot be chosen. The realizability check gets a PCC as input and returns a list of those options which can never be chosen due to conflicts in the PCC. The objective here is to fix the problems with those options to finally get an empty list as all options are realizable.

This problem can be solved by using *lemmas*.

Definition 3.1 (Lemma)**.** A *lemma* is a valid propositional logic formula which only contains variables already used in previous clauses. ▲

Hence, building the conjunction of a formula with a lemma does not effect the satisfiability of the formula.

As part of this thesis, we elaborated an algorithm on top of state-of-the-art SAT solving that generates lemmas. This algorithm is explained in detail in Chapter 6 on page 23.

In our context the generated lemmas state the reason for assigning certain values to variables and are implications of the form

$$Reason \Rightarrow variable \tag{3.1}$$

or

$$Reason \Rightarrow \neg variable \tag{3.2}$$

where $Reason$ is a subset of the set of clauses given as input in CNF. The first lemma in Equation (3.1) indicates that in all satisfying assignments $variable$ has the value `true` whereas the second lemma in Equation (3.2) indicates the value `false`.

We use these lemmas to compute those options which cannot be chosen and therefore are not *realizable*.

Example 3.11 (Realizability checking)**.** Consider the following PCC:

$$O_1 \Rightarrow O_2 \land O_3$$
$$O_5 \Rightarrow \neg O_4$$
$$O_1$$
$$O_5$$

The realizability check would give us the following lemmas:

$$\begin{aligned}(O_1) &\Rightarrow O_1\\(O_5) &\Rightarrow O_5\\(\neg O_1 \vee O_2) \wedge (O_1) &\Rightarrow O_2\\(\neg O_1 \vee O_3) \wedge (O_1) &\Rightarrow O_3\\(\neg O_5 \vee \neg O_4) \wedge (O_5) &\Rightarrow \neg O_4\end{aligned}$$

Hence, the option O_4 is not realizable whereas the options O_1, O_2, O_3 and O_5 have to be chosen in all configurations. ⧫

3.4.3 `VarGen` checking

The third problem is the so called `VarGen` (*Variance Generation*) check. The problem to solve here is to find the correct amount of hard- and software components which have to be designed for the production of some realizable solutions. Correct amount in this context means that there is no under- or over-engineering, i. e., all needed components are designed but no unnecessary ones.

We get as input a set of features of a specific `PCC`. The task now is to compute all combinations of realizable options which are consistent with the `PCC`. These combinations then have to be engineered. For example we have the option of having a dash cam and another option of having a GPS navigation system. In case of choosing both dash cam and navigation system it would be better to integrate both systems into one component instead of having two separate devices. The solution of the `VarGen` check would indicate which combinations of options are possible. This helps the engineer identifying the components which have to be developed and which components are not realizable. In our case the engineer would have to design a component combining a dash cam with a navigation system in addition to a standalone dash cam and a single navigation system.

The `VarGen` checking can be reduced to the problem of *All-SAT*. Instead of giving one satisfiable solution as is common for SAT solving the task of All-SAT is to compute all satisfiable solutions for a problem. In our case we gain all possible combinations of all options which satisfy the rules in the current `PCC`. If we restrict ourselves to certain *relevant* variables, e. g., dash cam and navigation system, we can solve the given `VarGen` task by computing the All-SAT solution.

Example 3.12 (VarGen checking)**.** Consider the following `PCC`:

$$O_1 \Rightarrow O_2 \wedge O_3$$

The given options are O_1, O_2. The All-SAT check would yield the following solutions:

$$\{\{O_1, O_2\}, \{\neg O_1, O_2\}, \{\neg O_1, \neg O_2\}\}$$

Here we see that of all four possible combinations of O_1 and O_2 the combination $O_1 \wedge \neg O_2$ is not possible. Therefore we would not have to consider this combinations in the following steps. ⧫

A detailed explanation on this topic can be found in Chapter 7 on page 31.

4 SAT solving

In this chapter we introduce the SAT solving algorithm which is used to solve our problems stated in Chapter 3 on page 7. The SAT solver gets a propositional logic formula in CNF as input and tries to find a *satisfying assignment*, i. e., an assignment for all occurring variables to Boolean values which satisfies the input CNF. If such an assignment cannot be found the given formula is *unsatisfiable* and an *unsatisfiable core* is returned, i. e., a subset of the clauses in the formula which is still unsatisfiable. This allows the user to find an explanation why the given formula is not satisfiable.

Typically SAT solvers are used as so called *black boxes* where it is not relevant to know the exact way the black box solves the problems. But as we like to extend the existing algorithm to better suit our needs we have to understand the SAT solving process in greater detail. Thus, this chapter gives an introduction to the *Davis-Putnam-Logemann-Loveland (DPLL) algorithm* for SAT solving [DLL62].

We consider the following running example during this chapter.

Example 4.1 (Problem)**.** Consider the following formula in CNF with 4 variables $x_1, \ldots, x_4$ and 5 clauses $c_1, \ldots, c_5$:

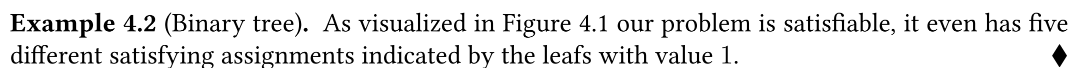
$$
\begin{aligned}
c_1 &: (x_1 \vee x_2 \vee x_4) \\
c_2 &: (x_2 \vee \neg x_4) \\
c_3 &: (x_3 \vee x_4) \\
c_4 &: (x_3 \vee \neg x_4) \\
c_5 &: (x_1 \vee x_2)
\end{aligned}
$$

We are interested in a solution to the problem:

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5$$

4.1 Enumeration

One of the oldest methods of solving SAT problems is *enumeration*. As the name indicates, enumeration is some sort of brute-force approach where all possible assignments of variables are tested one after the other. This process can be visualized with the help of a binary tree. In this tree the nodes correspond to (partial) assignments of the variables. The tree is constructed according to a variable ordering, i. e., the levels in the tree correspond to the different variables sorted by their ordering. The leafs indicate the satisfiability of the formulas considering the assignment of the variables along the path to this leaf. An illustration is given by the example in Figure 4.1.



Example 4.2 (Binary tree)**.** As visualized in Figure 4.1 our problem is satisfiable, it even has five different satisfying assignments indicated by the leafs with value 1. ♦

By constructing the binary tree we enumerate all the different variable assignments. If we encounter a satisfiable assignment we can stop and return *satisfiable* (`SAT`). Otherwise the problem is *unsatisfiable* (`UNSAT`).

This idea of enumeration is later used in the DPLL framework as well. There it occurs during the decisions for variables.

4.2 Resolution

An alternative solving method to enumeration is resolution which can be applied to a set of clauses. Thus, here we need a formula in CNF whereas enumeration works for general propositional logic formulas.

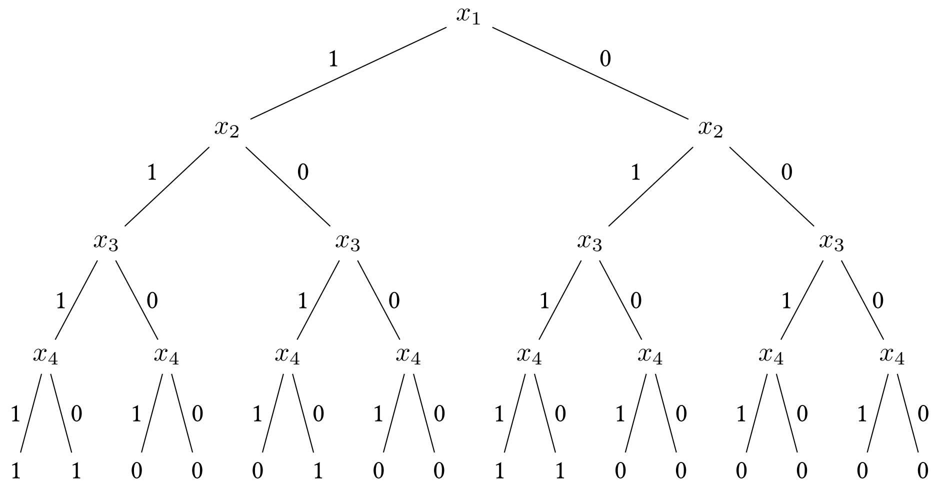


Figure 4.1: Binary tree for Example 4.1 with variable ordering $x_1 > x_2 > x_3 > x_4$.

Definition 4.1 (Resolution)**.** The binary *resolution* can be described by the following inference rule:

$$\frac{(a_1 \vee \cdots \vee a_n \vee \beta) \qquad (b_1 \vee \cdots \vee b_m \vee \neg\beta)}{(a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m)}$$

with $a_1, \ldots, a_n, b_1, \ldots, b_m$ being literals and β a variable.

If we have two *resolving clauses* $(a_1 \vee \cdots \vee a_n \vee \beta)$ and $(b_1 \vee \cdots \vee b_m \vee \neg\beta)$ where the *resolution variable* β is positive in one clause and negative in the other we can apply the resolution rule and get the *resolvent clause* $(a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m)$. ▲

This resolution rule can be used for computing the unsatisfiability of a CNF formula.

Theorem 4.1 (Resolution for unsatisfiability)**.** *A CNF formula is unsatisfiable iff there exists a finite series of binary resolution steps which derive the empty clause* $()$.

Proof. see [Rob65] □

Thus, resolution is refutation-complete, i. e., if a CNF formula is unsatisfiable we can always derive the empty clause and therefore prove its unsatisfiability.

Example 4.3 (Resolution)**.** Consider the given formula in CNF:

$$c_1 : (x_1 \vee x_2 \vee x_4)$$
$$c_2 : (x_2 \vee \neg x_4)$$
$$c_3 : (x_1 \vee \neg x_2)$$
$$c_4 : (\neg x_1)$$

Using resolution we get the derivation in Figure 4.2. We start with the clauses on the top level and repeatedly apply the resolution rule. The predecessors of a node indicate the resolving clauses and the node indicates the resolvent clause. As we can derive the empty clause the given formula is unsatisfiable. ♦

Since for each clause set, the number of clauses derivable by resolution is finite, resolution gives us a complete decision procedure.

4.3 DPLL

The following explanation of the DPLL framework is based on [KS08].

Combining enumeration and resolution we get the DPLL algorithm [DLL62]. Here we use enumeration for assigning variables and trying to find a satisfiable assignment. In case of a conflict,

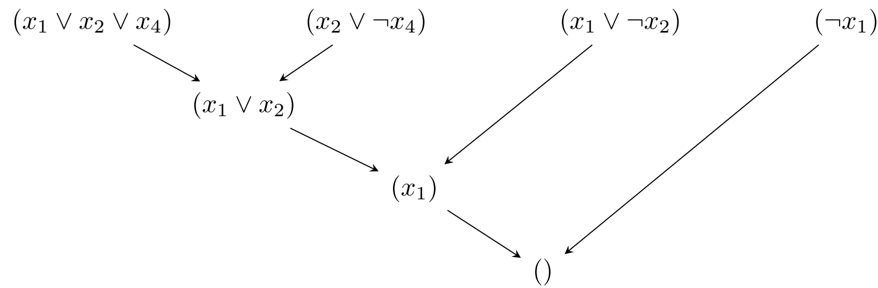


Figure 4.2: Derivation tree of the empty clause for Example 4.3

we use resolution to derive a clause which excludes similar conflicting assignments in the further search.

First we give an overview of the DPLL algorithm as shown in Algorithm 1 before explaining the concrete steps in detail.

Algorithm 1 DPLL algorithm

Input: propositional CNF formula φ
Output: SAT if φ is satisfiable, UNSAT otherwise

```
DPLL_SAT(φ)
begin
    conflict = BCP()                                            (1)
    if conflict ≠ null then                                     (2)
        return UNSAT                                            (3)
    end if                                                      (4)
    while true do                                               (5)
        if ¬DECIDE() then                                       (6)
            return SAT                                          (7)
        end if                                                  (8)
        conflict = BCP()                                        (9)
        while conflict ≠ null do                                (10)
            (backtrack-level, conflict') := ANALYZE_CONFLICT(conflict)   (11)
            if backtrack-level < 0 then                         (12)
                return UNSAT                                    (13)
            else                                                (14)
                BACKTRACK(backtrack-level)                      (15)
            end if                                              (16)
            conflict = BCP()                                    (17)
        end while                                               (18)
    end while                                                   (19)
end
```

The first step in the DPLL algorithm checks the consequences of unary clauses which directly imply assignments (Line 1). The function BCP() is described in Algorithm 2 below. If we already encounter a conflict we return UNSAT (Line 3).

Otherwise, in the main loop (Lines 5 to 19) we first apply enumeration and try to assign a value to a yet unassigned variable (Line 6). If all variables are already assigned, DECIDE() returns false and we have found a satisfying assignment. Then we can return SAT (Line 7). Otherwise, the consequences of the assignment are checked (Line 9) and, in case of a conflict, the conflict gets analyzed (Line 11). The conflict analysis procedure, discussed in Section 4.3.3 on page 17, might detect unsatisfiability (Line 13). Otherwise we propagate the changes from the possible conflict (Line 17) and finally perform a new enumeration step.

4.3.1 Boolean constraint propagation

After one decision we might have implications which determine the values of other variables in all satisfying extensions of the current partial assignment. These conclusions can be achieved by analyzing the clauses.

Definition 4.2 (States of a clause under an assignment)**.** A clause is in one of four states under an assignment:

- A clause is *satisfied* if one or more of its literals are assigned and satisfied.
- A clause is *conflicting* if all of its literals are assigned and not satisfied.
- A clause is *unit* if it is not satisfied and all but one literals are assigned.
- A clause is *unresolved* otherwise. ▲

One can easily see that unit clauses induce a variable assignment to satisfy this clause. This is called the *unit clause rule*.

Definition 4.3 (Unit clause rule)**.** Let $\mathcal{C}$ be a unit clause and l its unassigned literal. Then l is implied by $\mathcal{C}$ under the current assignment and we denote $\mathcal{C}$ as the *antecedent clause* of l

$$Antecedent(l) = \mathcal{C}$$ ▲

The algorithm of *Boolean Constraint Propagation (BCP)* is given in Algorithm 2. It repeatedly tries to apply the unit clause rule to assign all implied variables (Lines 6 to 13). It stops as soon as the assignment makes a clause conflicting and returns this clause (Line 3), or if there are no unit clauses under the current assignment it returns `null` (Line 15).

Algorithm 2 BCP algorithm

Input:
Output: conflicting clause if one exists, `null` otherwise

```
BCP()
begin
    while true do                                   (1)
        if existsConflictingClause() then           (2)
            return getConflictingClause()           (3)
        end if                                      (4)
        if existsUnitClause() then                  (5)
            clause = firstUnitClause()              (6)
            lit = unassignedLiteral(clause)         (7)
            var = variable(lit)                     (8)
            if negated(lit) then                    (9)
                assign(var, false)                  (10)
            else                                    (11)
                assign(var, true)                   (12)
            end if                                  (13)
        else                                        (14)
            return null                             (15)
        end if                                      (16)
    end while                                       (17)
end
```

4.3.2 Enumeration

The enumeration in the DPLL algorithm (Algorithm 1) takes places in the function `DECIDE`() (Line 6). `DECIDE`() choses a variable according to some heuristic and assigns either the value `true`

or `false`. Possible heuristics for selecting a variable and a value can be found in [MS99], e. g., *dynamic largest individual sum (DLIS)* which chooses the literal that satisfies the largest number of currently unsatisfied clauses, or *variable state independent decaying sum (VSIDS)*. VSIDS [Mos+01] introduces a counter for every literal which is incremented when this literal occurs in a newly added clause. Periodically all counters are decreased.

Additionally each decision is associated with a *decision level*. Intuitively the decision level is the depth in the binary decision tree. Decision levels improve the backtracking capabilities as we can undo previous decision up to a specific level but keep the remaining decisions instead of discarding all previous decisions.

4.3.3 Conflict analysis

A *conflict* with a conflicting clause c occurs during BCP if every literal in clause c is assigned the value `false`. We now know that at least one of the previous decisions lead to this clause being unsatisfied. Therefore we have to undo at least one assignment. This can be done by a two-stage approach. First we compute a so called *conflict clause* which captures a variable assignment responsible for the current conflict. Secondly we *backtrack* to a lower decision level and undo all decisions at higher decision levels.

The algorithm `ANALYZE_CONFLICT`() is given in Algorithm 3.

Algorithm 3 Conflict analysis algorithm

Input: cl: conflicting clause
Output: pair (backtrack level, asserting conflict clause)

```
ANALYZE_CONFLICT(cl)
begin
    dl = getCurrentDecisionLevel()                              (1)
    while (dl > 0 ∧ cl is not asserting) ∨ (dl == 0 ∧ cl ≠ () ) do   (2)
        l = last assigned literal in cl                          (3)
        a = antecedent(l)                                        (4)
        cl = resolvent(cl, a, variable(lit))                     (5)
    end while                                                    (6)
    addClause(cl)                                                (7)
    if |cl| == 1 then                                            (8)
        return (0, cl)                                           (9)
    else if |cl| == 0 then                                       (10)
        return (-1, cl)                                          (11)
    else                                                         (12)
        bt = second largest decision level in cl                 (13)
        return (bt, cl)                                          (14)
    end if                                                       (15)
end
```

We apply the resolution rule as explained in Section 4.2 on page 13 on the current clause and the antecedent of the last assigned literal in the clause and get the next clause (Lines 3 to 5). This is repeated until we either get an asserting clause or, if we are on decision level 0, we gain the empty clause $()$. Then we add this conflict clause to the set of all clauses (Line 7).

Next we determine the corresponding backtrack level. The rule here is to backtrack to the second most recent decision level in the conflict clause (Line 13), i. e., the highest level in the clause other than the current decision level. If all literals are assigned at the same decision level, the remaining clause has only one literal and the backtrack level is set to 0 (Line 9). If we are already on decision level 0 we cannot backtrack anymore and have found proof for the unsatisfiability by resolving the empty clause. This is indicated by a negative backtrack level (Line 11).

As the conflict clause is derived from the existing clauses, adding it to the clause set does not affect satisfiability. It is merely *learned* as it excludes a partial assignment which led to the current conflict. Thereby we ensure that similar conflicts do not occur during the further search. Further on we might exclude some more conflicting assignments which were not yet considered. In an intuitive way the solver uses conflict clauses to “learn from its mistakes”.

In the DPLL algorithm in Algorithm 1 on page 15 a negative backtrack-level indicates that the conflict does not arise from previous decisions but from inherent variable assignments and therefore the formula is UNSAT (Line 13 on page 15).

Otherwise, after learning the conflict clause we try to undo some recent decisions. However, we do not want to backtrack too far as we would have to redo some work. The concrete backtrack-level is returned from ANALYZE_CONFLICT() (Line 11 on page 15). By backtracking to a certain backtrack-level (Line 15 on page 15) we undo all decisions taken after this level. Further on all corresponding implications from these decisions are erased as well.

It is important to recognize that after backtracking, the asserting conflict clause is unit, i. e., it leads to a new implication, which again needs to be propagated (Line 17 on page 15).

4.3.4 Complete example

We conclude this chapter by giving a complete run of the DPLL algorithm on the previous problem of Example 4.1 on page 13. The clauses were:

$$c_1 : (x_1 \vee x_2 \vee x_4)$$
$$c_2 : (x_2 \vee \neg x_4)$$
$$c_3 : (x_3 \vee x_4)$$
$$c_4 : (x_3 \vee \neg x_4)$$
$$c_5 : (x_1 \vee x_2)$$

Example 4.4 (Complete DPLL run)**.** Initially, there are no assignments made at decision level 0. Assume that the heuristic in DECIDE() chooses to assign x_1 to false. This first assignment indicates the first decision at decision level 1 written as

$$\neg x_1@1$$

Then clause c_5 becomes a unit clause. Using the unit clause rule we deduce a new variable assignment $x_2 = \texttt{true}$ written as:

$$x_2@1$$

Now clauses c_1, c_2 and c_5 are satisfied.
Next we set the assignment for the following variable x_3 as:

$$\neg x_3@2$$

Now clauses c_3 and c_4 are unit clauses. First we consider clause c_3 in BCP and get the following assignment:

$$x_4@2$$

However, the clause c_4 is now conflicting as x_3 and $\neg x_4$ are both false. Our next step is to resolve this conflict. We get the following conflict clause c_6 from the conflict analysis as the resolvent of c_3 and c_4:

$$c_6 = (x_3)$$

As the only decision level in the conflict clause is 2, which is the current decision level, we backtrack to the lowest level 0, i. e., all decisions are erased. Now, c_6 implies the assignment

$$x_3@0$$

Then the clauses c_3 and c_4 are satisfied.
Our next decision is

$$\neg x_1@1$$

This leads to clause c_5 being a unit clause and the assignment

$$x_2@1$$

Now all clauses are satisfied and the CNF formula is satisfiable. Making a further decision for the remaining variable x_4 as

$$\neg x_4@2$$

we get a satisfying assignment as

$$\{\neg x_1, x_2, x_3, \neg x_4\}$$

5 Incremental SAT solving

After introducing the general idea behind SAT solving we now come to the actual solving of our three main problems. Here we start with the *order checking* as explained in Section 3.4.1 on page 10.

As stated previously for order checking we get an order with its chosen options and a corresponding PCC. We encode the options and the PCC as propositional logic formulas and solve them with a SAT solver. The solver gives us the answer whether the order is satisfiable or unsatisfiable.

We could be content with this approach as we can solve our problem. Nevertheless there is room for improving and accelerating the solving process. Our main approach here is to exploit *incrementality* in SAT solving.

A solver is *incremental* if it reuses results from the previous checks when checking again after *removing* and *adding* clauses.

The first check is performed on a set of clauses $\mathcal{C}$. Afterwards this problem is changed by removing clauses $\mathcal{R}$ and adding clauses $\mathcal{A}$ which leads to the new set of problem clauses

$$\mathcal{C}' = (\mathcal{C} \setminus \mathcal{R}) \cup \mathcal{A}.$$

The check on $\mathcal{C}'$ now reuses results from the check on $\mathcal{C}$ instead of discarding them and starting anew.

Using the incrementality of the SAT solver we can take advantage of the previous computations as we do not have to discard them completely when changing the formulas to solve. Instead the solver tries to preserve as much useful gained knowledge as possible, i. e., activities of variables for decision heuristics, certain conflict clauses, etc.. This helps accelerating the solving process as the incremental solving is faster than multiple isolated checks.

5.1 Interfaces

The incrementality in the SAT solver is achieved by calling the interfaces as proposed in the *SMT-LIB* file format [BST10]. Here we mainly use the commands `push` and `pop` which work on a stack. Calling `push` introduces a new layer on the stack on which all newly added formulas are put. Calling `pop` then removes all formulas up to the highest `push` layer.

Example 5.1 (Push and pop). A visualization of `push` and `pop` is shown in Figure 5.1.

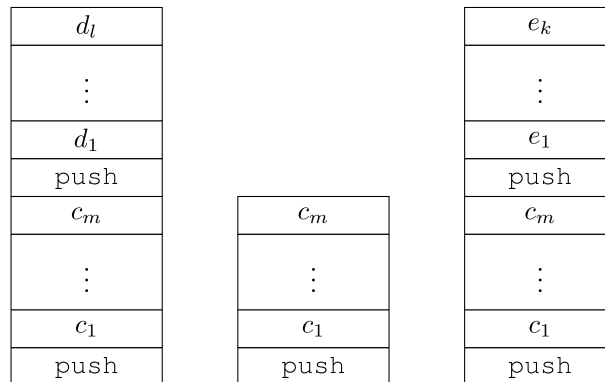


Figure 5.1: Push and pop

The left one is the stack after adding clauses $c_1, \ldots, c_m$, calling `push` and then adding additional clauses $d_1, \ldots, d_l$. The middle stack is after calling `pop`. The right stack is after calling `push` again and adding new clauses $e_1, \ldots, e_k$. ♦

5.2 Order checking

Incrementality is beneficially for bulk checking. Each order is based on a product type, i. e., a PCC. As some orders share the same PCC we sort the orders by their corresponding PCC and start by loading the first PCC. Now for every order we call push and then add the options from the order as new clauses. After solving the problem we call pop and thus only remove those clauses corresponding to the options of the order. The clauses for the PCC remain untouched and all gained knowledge about them remains in the solver, e. g., transformed input in CNF, unary learnt clauses, accumulated values for the decision heuristic. Next we consider another order and add its options again. This process is repeated for all orders.

Example 5.2 (Order checking)**.** The workflow for order checking is visualized in Figure 5.2.

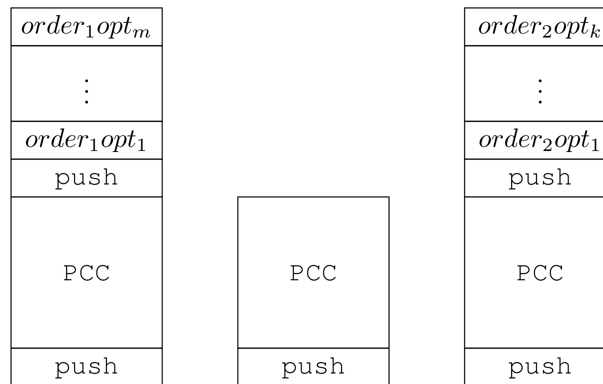


Figure 5.2: Workflow of order checking

As explained before the PCC is kept whereas the orders change during the process. ♦

The incrementality gives us speed improvements as we do not have to load the PCC for every order but each PCC only once during the bulk checking. Further on we keep possible knowledge of the specific PCC during the solving process which might again improve the solving performance.

6 Lemma generation

We now come to the second problem: the *realizability check* as stated in Section 3.4.2 on page 11. Here we check whether every option in the given PCC can be taken, i. e., every option is realizable, and whether there are options which always have to be taken.

6.1 Naive algorithm

The naive approach to this problem can be seen in Algorithm 4. Here we first add the given PCC (Line 4). Then each option is added (Line 7) and checked successively (Line 8). If the solver states that the PCC along with the current option is satisfiable, this option is realizable, otherwise it cannot be taken and is added to the list of unrealizable options (Line 9).

Algorithm 4 Naive algorithm for realizability check

Input: PCC
options in PCC
Output: list of all unrealizable options

```
REALIZABILITY_NAIVE(PCC, options)
begin
    optionsUnrealizable = ∅                                              (1)
    PCC_CNF = ToCNF(PCC)                                                 (2)
    push()                                                               (3)
    add(PCC_CNF)                                                         (4)
    for each option_i ∈ options do                                       (5)
        push()                                                           (6)
        add(option_i)                                                    (7)
        if check() = UNSAT then                                          (8)
            optionsUnrealizable = optionsUnrealizable ∪ {option_i}       (9)
        end if                                                           (10)
        pop()                                                            (11)
    end for                                                              (12)
    pop()                                                                (13)
    return optionsUnrealizable                                           (14)
end
```

In this solution we have to assert every option, call the check routine and then evaluate the result of the solver in an external wrapper. This workflow is shown in Figure 6.1.

6.2 Improved algorithm

The above approach suffices to solve the realizability problem, but we still can improve the time consumption.

The first improvement to the naive approach does not only use the UNSAT result of the solver but also the SAT result. In case of SAT in Line 8 we know that every chosen option in the satisfying assignment is realizable. Hence, we do not need to check these options anymore as we already know their realizability.

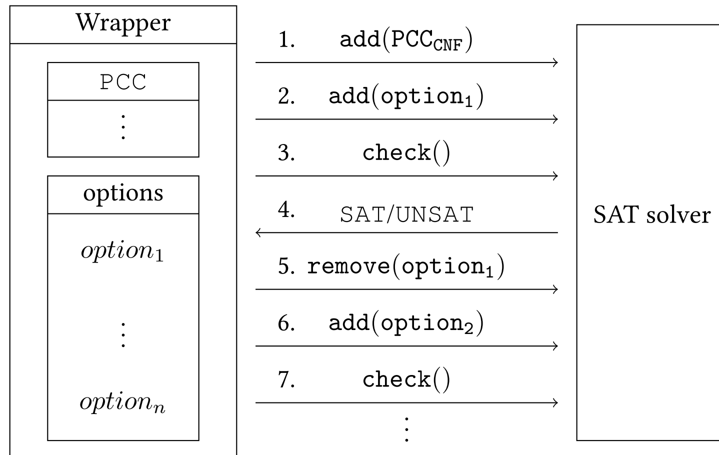


Figure 6.1: Naive workflow for the realizability check

Example 6.1 (Satisfying assignment)**.** Consider the following clauses:

$$c_1 : (x_1 \lor x_2)$$
$$c_2 : (x_2 \lor \neg x_4)$$
$$c_3 : (x_3 \lor x_4)$$
$$c_4 : (\neg x_1)$$

We get the following satisfying assignment after the solving:

$$\{\neg x_1, x_2, x_3, x_4\}$$

Now we know that not only variable x_2 but also x_3 and x_4 are satisfiable and therefore these variables do not need to be considered anymore. ♦

The improved algorithm can be seen in Algorithm 5, especially the added part in Lines 11 to 15 dealing with the SAT case.

6.3 Final algorithm

If we use the solver as a "black box" we do not have much room for improvement and a lot of communication has to happen between the solver and the wrapper. But we want to decrease the amount of communication between the two programs and autonomously compute as much as possible in the solver. Now we benefit from our own solver SMT-RAT as we can implement the needed behavior in the solver and therefore use the solver as a "white box" where we know how the solving takes places.

The final algorithm for solving the realizability abstracts from the specific setting of options to a more general approach of so called *relevant variables*. In SMT-RAT we can add and remove those relevant variables which then can be used for certain tasks depending on the used solver modules. In our case the relevant variables state those variables which should be used for the realizability check, i. e., the corresponding options.

The final algorithm makes use of *lemmas* (see Definition 3.1 on page 11). Lemmas represent inherent knowledge of the added formulas and do not change the satisfiability of the formulas. In the context of incrementality lemmas are used to remember gained information which then help to accelerate future checks.

In our setting, we use lemmas for encoding the reasons for specific variable assignments. Instead of only returning the set of unrealizable variables we return each variable with its corresponding reason implying the specific assignment. This helps the engineer identifying the reasons for the unrealizability of certain options and therefore improves the process of writing and modifying the set of rules.

Algorithm 5 Improved algorithm for realizability check

Input: `PCC`
options in `PCC`
Output: list of all unrealizable options

```
REALIZABILITY_IMPROVED(PCC, options)
begin
    optionsUnrealizable = ∅                                          (1)
    PCC_CNF = ToCNF(PCC)                                             (2)
    push()                                                           (3)
    add(PCC_CNF)                                                     (4)
    for each option_i ∈ options do                                   (5)
        push()                                                       (6)
        add(option_i)                                                (7)
        if check() = UNSAT then                                      (8)
            optionsUnrealizable = optionsUnrealizable ∪ {option_i}   (9)
        else                                                         (10)
            for each var ∈ assignment do                             (11)
                if var = true then                                   (12)
                    options = options \ {var}                        (13)
                end if                                               (14)
            end for                                                  (15)
        end if                                                       (16)
        pop()                                                        (17)
    end for                                                          (18)
    pop()                                                            (19)
    return optionsUnrealizable                                       (20)
end
```

In our problem case we *generate lemmas* during the solving process to capture implications of variable values. For each variable assignment at decision level 0, we remember which clauses are responsible for this assignment and store this reason implying the variable assignment as a lemma:

$$reason \Rightarrow variable \qquad \text{or}$$
$$reason \Rightarrow \neg variable$$

Example 6.2 (Lemma generation at decision level 0)**.** Assume we have the clauses from Example 6.1 on page 23. We then compute the following lemmas:

$$c_4 : (\neg x_1) \Rightarrow \neg x_1$$
$$c_1 : (x_1 \vee x_2) \wedge c_4 : (\neg x_1) \Rightarrow x_2$$ ♦

We restrict ourselves to assignments at decision level 0 before any decisions have taken place. All later lemmas additionally depend on the current variable decisions and therefore are not valid globally but only under these variable assignments.

However, using these lemmas does not suffice for solving the realizability as in general we do not capture all variables at decision level 0. Hence, we use the previous mentioned approach of successively assigning the remaining variables and checking the satisfiability. As we do this whole procedure inside the `SMT-RAT` solver we avoid most of the previous communications.

The final algorithm is based on the DPLL algorithm (see Algorithm 1 on page 15) and is stated in Algorithm 6.

As in the original DPLL algorithm we start by calling the `BCP`() algorithm in Line 2. If a conflict already occurred at decision level 0 the CNF formula φ is `UNSAT` and we return the reason for this unsatisfiability as a lemma (Line 4). The function `leaves`() collects all clauses responsible for the given conflict clause and is explained later.

Otherwise, we perform the computation of lemmas similar to the improved algorithm of Algorithm 5. We iterate over all relevant variables (Line 6) and in each iteration we assign the first

Algorithm 6 DPLL algorithm for realizability

```
Input: propositional CNF formula φ
       relevantVariables: set of all variables of interest
Output: set of lemmas indicating variable values

DPLL_REALIZABILITY(φ, relevantVariables)
begin
    lemmas = ∅                                                          (1)
    conflict = BCP()                                                    (2)
    if conflict ≠ null then                                             (3)
        return {leaves(conflict) ⇒ false}                               (4)
    end if                                                              (5)
    while relevantVariables ≠ ∅ do                                      (6)
        varCheck = relevantVariables.first()                            (7)
        relevantVariables = relevantVariables \ {varCheck}              (8)
        done = false                                                    (9)
        while ¬done do                                                  (10)
            if ¬DECIDE_REALIZABILITY(varCheck) then // SAT              (11)
                for each var ∈ assignment do                            (12)
                    if var = true then                                  (13)
                        relevantVariables = relevantVariables \ {var}   (14)
                    end if                                              (15)
                end for                                                 (16)
                done = true                                             (17)
            else                                                        (18)
                conflict = BCP()                                        (19)
                while conflict ≠ null do                                (20)
                    (backtrack-level, conflict′) = ANALYZE_CONFLICT(conflict)  (21)
                    if backtrack-level < 0 then // UNSAT                (22)
                        return {leaves(conflict′) ⇒ false}              (23)
                    else                                                (24)
                        BACKTRACK(backtrack-level)                      (25)
                    end if                                              (26)
                    conflict = BCP()                                    (27)
                end while                                               (28)
            end if                                                      (29)
        end while                                                       (30)
        BACKTRACK(0)                                                    (31)
    end while                                                           (32)
    for each lit ∈ assignment do                                        (33)
        lemmas = lemmas ∪ {leaves((lit)) ⇒lit}                          (34)
    end for                                                             (35)
    return lemmas                                                       (36)
end
```

variable to the value `true`. This in done in the modified function `DECIDE_REALIZABILITY`() (Line 11). The corresponding algorithm is stated in Algorithm 7.

Algorithm 7 Decision algorithm for realizability

Input: prioritized variable var
Output: `false` if all variables are already assigned, `true` otherwise

```
DECIDE_REALIZABILITY(var)
begin
    if var == unassigned then           (1)
        decisionLevel++                 (2)
        var = true                      (3)
        return true                     (4)
    else                                (5)
        return DECIDE()                 (6)
    end if                              (7)
end
```

The function `DECIDE_REALIZABILITY`() gets a variable as input and tries to assign this variable before any other variable. If the prioritized variable is currently not assigned we increase the decision level (Line 2) and assign this variable to `true` (Line 3). If the variable is already assigned we call the original decision procedure for all other variables (Line 6).

After assigning the current variable in Algorithm 6, we remove it from the relevant variables to ensures termination of the loop in Line 6 and perform a regular DPLL solving. After the solving the current assignment is reset by backtracking to decision level 0 (Line 31) so we can restart the solving in an initial state again. Especially the previously assigned variable is reset as we do not consider it anymore.

If the result of the DPLL solving is `SAT` (Lines 14 to 17), we remove all satisfied variables from the set of all relevant variables.

The interesting part happens, if we find a conflict. If the current decision level is 0, the conflict is independent of the current variable we assigned at decision level 1. This indicates that the given CNF formula is already unsatisfiable and we return the corresponding reason as a lemma (Line 23).

If the decision level is higher than 0 we backtrack as seen in the original algorithm. However, as we add the conflict clause to the set of all clauses, we can analyze those clauses later.

In particular, if the conflict occurs at decision level 1, this indicates that the assignment of the current relevant variable to `true` implies a conflict and this variable is not realizable. After backtracking the resulting asserting conflict clause implies a new assignment at decision level 0 of the relevant variable to `false` which captures the unrealizability of this variable.

After considering all relevant variables we additionally generate lemmas for all variable assignments at decision level 0 (Lines 33 to 35). As we backtracked before, decision level 0 is ensured and all assignments occurring are independent of variable decisions. The reason for each variable assignment is given by the unit clause which became unit because of possible previous assignments to variables in the same clause. Therefore we have to consider all input clauses responsible for making this clause unit.

This is done by the function `leaves`(). As we gained the given clause by performing resolution on clauses, we can collect the leaves of the resolution tree for the clause and get all input clauses which are responsible for the given clause. These clauses are the reason for the variable assignment and are added as a new lemma (Line 34). As explained before, the unrealizable variables are assigned to `false` at decision level 0 and therefore are captured by these lemmas.

The algorithms ends with returning all computed lemmas (Line 36).

Using the function `leaves`() we are able to give a reason why a formula is `UNSAT` as seen in Line 23. This reason is called *unsatisfiable core* [ZM03].

Definition 6.1 (Unsatisfiable core)**.** An *unsatisfiable core* of an unsatisfiable formula in CNF is any unsatisfiable subset of the original set of clauses. ▲

An unsatisfiable core should be as small as possible and allows for a better understanding why the formulas are unsatisfiable. Instead of considering all formulas only a small portion must be analyzed and is sufficient on its own to prove unsatisfiability.

In case of unsatisfiability in our realizability check we construct a new lemma with an unsatisfiable core as a reason for the unsatisfiability.

Example 6.3 (Unsatisfying assignment)**.** Consider the following clauses:

$$c_1 : (\neg x_1 \vee \neg x_2)$$
$$c_2 : (x_2 \vee \neg x_4)$$
$$c_3 : (x_3 \vee x_4)$$
$$c_4 : (x_1)$$

Assume we asserted x_2 which yields the result unsatisfiable. Then an unsatisfiable core is:

$$\{(x_2), c_4 = (x_1), c_1 = (\neg x_1 \vee \neg x_2)\}$$ ♦

As stated before, the lemma generation is not only necessary for our realizability check, but can be useful in general by improving the solving performance. On the other hand this lemma generation takes additional computation time, therefore it must be activated implicitly in `SMT-RAT`. This is done by setting a so called `LemmaLevel` which indicates the degree to which we want to compute additional lemmas.

6.3.1 Complete example

We conclude this chapter by giving an example of a realizability check.

Example 6.4 (Realizability check)**.** Consider the given clauses:

$$c_1 : (x_1 \vee x_2)$$
$$c_2 : (x_2 \vee \neg x_4)$$
$$c_3 : (\neg x_3 \vee x_4)$$
$$c_4 : (\neg x_1)$$
$$c_5 : (\neg x_3 \vee \neg x_4)$$

We start the realizability check with the list of relevant variables $\{x_3, x_4\}$ and first assert variable x_3. Then the result is `UNSAT` with the following unsatisfiable core.

$$\{(x_3), c_3 = (\neg x_3 \vee x_4), c_5 = (\neg x_3 \vee \neg x_4)\}$$

Performing resolution we get the following conflict clause:

$$c_6 : (\neg x_3)$$

This clause is added to the set of clauses.
Next we consider the remaining variable x_4 which yields the satisfying assignment:

$$\{\neg x_1, x_2, \neg x_3, x_4\}$$

As all relevant variables are checked we now compute the lemmas for decision level 0. The remaining variable assignments are:

$$\{\neg x_1, x_2, \neg x_3\}$$

The first variable assignment $\neg x_1$ was implied by unit clause c_4 and yields the lemma:

$$c_4 \Rightarrow \neg x_1$$

Variable x_2 has the resolution tree depicted in Figure 6.2. Therefore the assignment for variable x_2 is implied by clauses c_1 and c_4. The corresponding lemma is:

$$(c_1 \wedge c_4) \Rightarrow x_2$$

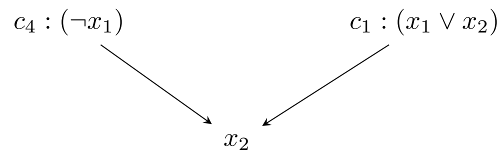


Figure 6.2: Resolution tree for x_2

As seen before the variable assignment $\neg x_3$ was implied by the conflict clause c_6 which was gained by conflict resolution on clauses c_3 and c_5. Therefore the corresponding lemma is:

$$(c_3 \wedge c_5) \Rightarrow \neg x_3$$

The algorithm terminates with the returned lemmas:

$$\begin{aligned} c_4 &\Rightarrow \neg x_1 \\ (c_1 \wedge c_4) &\Rightarrow x_2 \\ (c_3 \wedge c_5) &\Rightarrow \neg x_3 \end{aligned}$$

Analyzing the lemmas in the wrapper we could extract the information that options x_1 and x_3 are not realizable. Further on, option x_2 has to be chosen in all configurations. ♦

7 All-SAT

Last we consider the third problem: the `VarGen` check as introduced in Section 3.4.3 on page 12. Here we get a list of options and a list of `PCCs` as input and for each `PCC` we have to compute those combinations of options which are possible in this `PCC`.

7.1 Naive algorithm

The naive approach of solving this problem is to try all possible combinations of options and check each of them for satisfiability.

Example 7.1 (All combinations)**.** Assume we have the options x_1, x_2, x_3. Then all possible combinations would be:

$$\begin{aligned} \{\{\ x_1,\ \ x_2,\ \ x_3\},& \qquad \{\ x_1,\ \ x_2, \neg x_3\},\\ \{\ x_1, \neg x_2,\ \ x_3\},& \qquad \{\ x_1, \neg x_2, \neg x_3\},\\ \{\neg x_1,\ \ x_2,\ \ x_3\},& \qquad \{\neg x_1,\ \ x_2, \neg x_3\},\\ \{\neg x_1, \neg x_2,\ \ x_3\},& \qquad \{\neg x_1, \neg x_2, \neg x_3\}\ \} \end{aligned}$$ ♦

This approach of iterating over all combinations is formalized in Algorithm 8.

Algorithm 8 Naive algorithm for `VarGen` check

Input: `PCC`
options
Output: set of all realizable combinations of options

```
VARGEN_NAIVE(PCC, options)
begin
    possibleOptions = ∅                                          (1)
    PCC_CNF = ToCNF(PCC)                                         (2)
    push()                                                       (3)
    add(PCC_CNF)                                                 (4)
    combinations = 2^options                                     (5)
    for each combination_i ∈ combinations do                     (6)
        push()                                                   (7)
        for each option_j ∈ options do                           (8)
            if option_j ∈ combination_i then                     (9)
                assign(option_j)                                 (10)
            else                                                 (11)
                assign(¬ option_j)                               (12)
            end if                                               (13)
        end for                                                  (14)
        if check() = SAT then                                    (15)
            possibleOptions = possibleOptions ∪ {combination_i}  (16)
        end if                                                   (17)
        pop()                                                    (18)
    end for                                                      (19)
    pop()                                                        (20)
    return possibleOptions                                       (21)
end
```

As before we start by adding the PCC (Line 4). Next we compute a list of all possible combinations of options (Line 5) and iterate over all those combinations (Line 6). For each combination its options are added (Lines 8 to 14) and then we check the satisfiability (Line 15). If the result is SAT, we have found another possible combination of options (Line 16).

7.2 All-SAT

It is easy to see that the above approach lacks in performance as we need $2^{\#options}$ checks which is exponential in the number of given options. We want to solve the whole problem in one solver call instead of iteratively communicating each tuple of options and checking it. Therefore we can make use of the solver SMT-RAT and extend it to support the computation of all satisfying assignments for a given set of Boolean variables.

Our problem of computing all possible combinations is similar to the problem of computing all satisfying solutions for given formulas. This problem is called All-SAT.

Definition 7.1 (All-SAT)**.** The task of *All-SAT* is to find all satisfying solutions to a given SAT problem. ▲

By extending the solver SMT-RAT to solve the All-SAT problem we can solve the VarGen check as well. But we are not interested in all assigned variables but only in some *relevant variables*. Therefore we can improve our solving to focus on these variables.

7.2.1 Final algorithm

The implemented algorithm for All-SAT is based on the DPLL algorithm (see Algorithm 1 on page 15) and is outlined in Algorithm 9. Here we once again use the *relevant variables* which now capture those variables constituting the possible combinations.

We start by solving the SAT problem as usual. If the problem is UNSAT immediately, we are done (Line 4). Otherwise, if the solver finds a satisfying assignment (Line 9), we project this assignment to the set of relevant variables (Line 10) to only keep those variables, we are interested in. Next we have to ensure that we do not find the current solution again in a later run. This is assured by constructing an excluding clause (Lines 12 to 20) which is a negation of the current assignment. This clause is similar to the conflict clauses in the original DPLL algorithm and excludes the current solution from all future checks.

After finding a solution we continue solving to find another solution. If we do not find any further solution, i. e., the result is UNSAT and we return the set of all solutions (Line 27).

Using this approach we can compute all possible combinations for a given PCC and a set of options. Applying this All-SAT algorithm iteratively for all PCCs we solve the VarGen problem.

7.2.2 Complete example

This chapter concludes with a complete example of a VarGen check.

Example 7.2 (VarGen check)**.** Let $\{x_2, x_4\}$ be the relevant variables and the given clauses are:

$$c_1 : (x_1 \vee x_2)$$
$$c_2 : (x_2 \vee \neg x_4)$$
$$c_3 : (\neg x_3 \vee x_4)$$

The first satisfying assignment is:

$$\{x_1, x_2, \neg x_3, x_4\}$$

which gives the first solution:

$$\{x_2, x_4\}$$

The corresponding excluding clause is:

$$c_4 : \ (\neg x_2 \vee \neg x_4)$$

Algorithm 9 DPLL algorithm for All-SAT

Input: propositional CNF formula φ
relevantVariables: set of Boolean variables which are relevant for All-SAT
Output: all satisfying assignments of the relevant variables

```
DPLL_ALL_SAT(φ, relevantVariables)
begin
    solutions = ∅                                                        (1)
    conflict = BCP()                                                     (2)
    if conflict ≠ null then                                              (3)
        return ∅                                                         (4)
    end if                                                               (5)
    while true do                                                        (6)
        done = false                                                     (7)
        while ¬done do                                                   (8)
            if ¬DECIDE() then // SAT                                     (9)
                solution = assignment|relevantVariables                  (10)
                solutions = solutions ∪ {solution}                       (11)
                excludeClause = false                                    (12)
                for each var ∈ relevantVariables do                      (13)
                    if var = true then                                   (14)
                        excludeClause = excludeClause ∨ ¬ var            (15)
                    else                                                 (16)
                        excludeClause = excludeClause ∨ var              (17)
                    end if                                               (18)
                end for                                                  (19)
                φ = φ ∪ excludeClause                                    (20)
                done = true                                              (21)
            else                                                         (22)
                conflict = BCP()                                         (23)
                while conflict ≠ null do                                 (24)
                    (backtrack-level, conflict′) := ANALYZE_CONFLICT(conflict)   (25)
                    if backtrack-level < 0 then  // UNSAT                (26)
                        return solutions                                 (27)
                    else                                                 (28)
                        BACKTRACK(backtrack-level)                       (29)
                    end if                                               (30)
                    conflict = BCP()                                     (31)
                end while                                                (32)
            end if                                                       (33)
        end while                                                        (34)
        BACKTRACK(0)                                                     (35)
    end while                                                            (36)
end
```

Then we search for another solution and get the satisfying assignment:

$$\{x_1, x_2, \neg x_3, \neg x_4\}$$

The next solution is:

$$\{x_2, \neg x_4\}$$

We add the excluding clause:

$$c_5 : \ (\neg x_2 \vee x_4)$$

Searching for another solution we find the assignment:

$$\{x_1, \neg x_2, \neg x_3, \neg x_4\}$$

Therefore the third solution is:

$$\{\neg x_2, \neg x_4\}$$

This implies the excluding clause:

$$c_6 : \ (x_2 \vee x_4)$$

Now the solver does not find further solutions. Therefore we return all three solutions:

$$\{\{x_2, x_4\}, \{x_2, \neg x_4\}, \{\neg x_2, \neg x_4\}\}$$

Furthermore, we can see that the combination $\{\neg x_2, x_4\}$ is not possible, because it contradicts the clause $c_2 : (x_2 \vee \neg x_4)$. ♦

8 Implementation

In this chapter we describe how we implemented the previously described algorithms. We start by giving an overview of the used architecture.

8.1 Architecture

The used architecture is visualized in Figure 8.1.

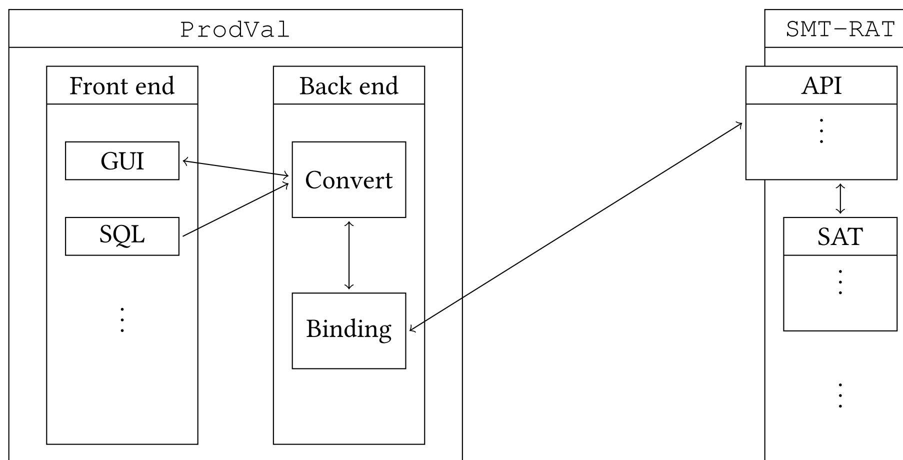


Figure 8.1: Architecture

The architecture consists of two main parts. The first part is called ProdVal (*Product Validator*) and contains all code related to input data from the company, e. g., the PCCs and their propositional encoding. The second part is our solver SMT-RAT [Cor+15a] [Cor+15b] [Cor+12] which performs the actual solving. Our implementation also offers the possibility to exchange the solver to use for example Z3 [Mic15] [DMB08]. However, using SMT-RAT we benefit from our custom implementations for realizability and VarGen check which are not available in Z3.

We start by explaining the ProdVal tool. This tool is written in C# and is divided into a front end and a back end. The front end was developed by the company. It handles the web-based GUI (see Figure 8.2) and the storage of the PCCs in a database. All user interaction is handled here and only the concrete tasks are forwarded to the back end. The back end gets the PCC rules, the current task and possibly additional data as input from the front end and then has to solve the given task by using a SAT solver. We can choose from two possible back ends: the company back end using Z3 and our back end using SMT-RAT.

In our back end we start by converting the given PCC rules into the *SMT-LIB* format [BST10]. This transformation was explained in detail in Section 3.2 on page 7. Particularly, we have to encode the *validity period* of the rules as explained in Example 3.7 on page 9 and the relationship of *exactly one* for features as seen in Example 3.2 on page 7.

After encoding all rules we communicate them to the solver SMT-RAT.

8.2 SMT-RAT

For SAT solving we use our own solver SMT-RAT. SMT-RAT consists of multiple modules which can be used according to a user-defined strategy to solve SMT problems. In our case we restrict ourselves to SAT problems and therefore mostly use the so called SATModule. This module is based on the solver MiniSat [ES03]. The custom changes necessary for lemma generation and

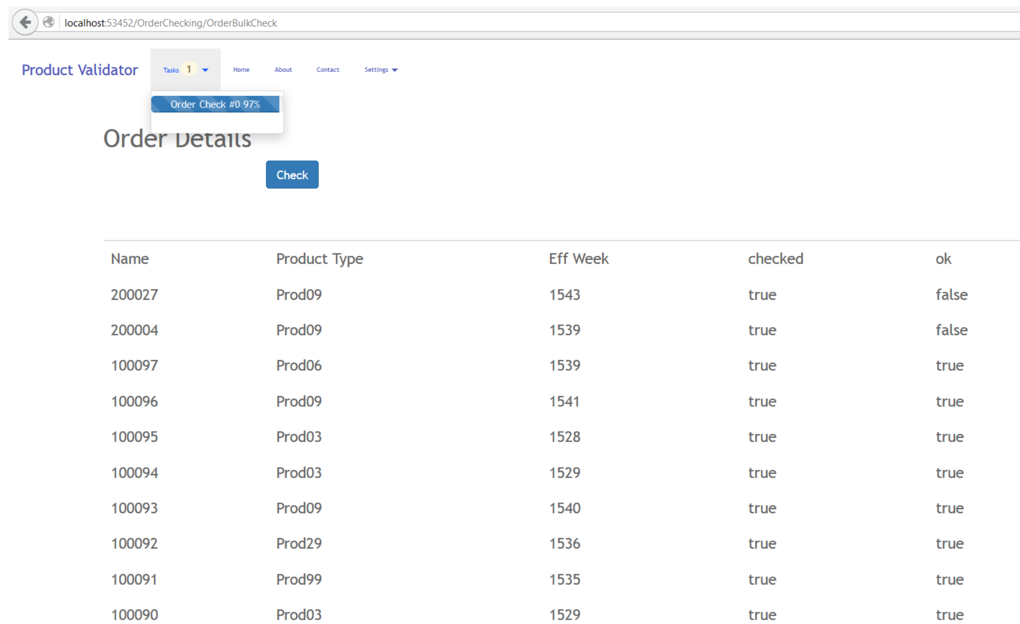


Figure 8.2: A screenshot of the GUI

All-SAT were all implemented in this module. Furthermore, we increase the solving performance by first applying the `PreprocessingModule` which simplifies the input formula.

When starting with the project the main problem in the implementation was the multi platform compatibility. Previously `SMT-RAT` was developed only for `Linux` but for the usage in the company the solver also must be running on `Windows`. Thus, porting `SMT-RAT` to `Windows` was the first and very time consuming task.

On `Linux` we use the library `GMP` [Gra15] for numeric operations, but it does not work on `Windows`. Luckily on `Windows` we can use the library `MPIR` [GH15] which is a fork of `GMP`. Additionally, a lot of problems on `Windows` relate to the compiler of `Microsoft Visual Studio` which does not fully support the `C++` standard in version 11. We solved these problems by distinguishing between the different compilers and changing parts of the source code depending on the used compiler.

For integration with `ProdVal` we defined a custom API for `SMT-RAT` and included the solver as a *dynamic-link library (DLL)* which can be called from `C#`. The main commands for a SMT solver (adding and removing formulas, `push`, `pop`, `check`) are available here. The data exchange is mostly done via string representation as this is most flexible and does not depend on custom data structures which are not available in all used programming languages. Rules can be added in their usual encoding as A `IMPL` B and are transformed into CNF in `SMT-RAT`.

In `SMT-RAT` the order check can be solved by simply calling `check` which returns if the given set of rules and asserted options is satisfiable.

As explained in Chapter 6 on page 23 the lemma generation can be activated with setting the `LemmaLevel` to `ADVANCED` to only use this time consuming calculation when needed, e. g., in the realizability check. With `addRelevantVariable()` all variables we are interested in can be set. After checking the `PCC`, the solver can return the `impliedVariables` which contain all relevant variables which either never can be satisfied or are always satisfied.

Last when calling `allModels()` after a `check` the solver computes all satisfying solutions consisting of the relevant variables instead of returning just one solution. This helps solving the `VarGen` task.

9 Evaluation

We compared our approach with SMT-RAT to the second similar implementation from the company which uses Z3 as a solver. Unfortunately it was not possible to compare our results to the currently used solution which is not based on SAT solver technology. But as the approach with Z3 already performs better than the existing solution, this indicates that a replacement of the current software is beneficial.

All evaluations were executed on an Intel(R) Core(TM) i5-4200U CPU with 1.60 GHz, 12 GB RAM and Windows 8.

9.1 Order checking

First we evaluate the order checking which is a simple SAT checking. Here we got a sample from the company with 103 different orders, where 6 of them were unsatisfiable. Due to business secrets it was not possible to perform the check on the actual data of up to 50,000 orders.

The results can be seen in Table 9.1.

Sorted	Solver	Load.	Prep.	Check	Total
✗	SMT-RAT	10.01	28.18	4.42	32.60
	Z3	9.57	23.48	4.53	28.01
✓	SMT-RAT	9.85	12.82	4.45	17.26
	Z3	9.67	24.11	4.48	28.60

Table 9.1: Order checking

The first column indicates whether the orders were presorted and the second column states the used solver SMT-RAT or Z3. The column *Load.* indicates the time in seconds need for loading the rules from the database, the column *Prep.* indicates the time needed for preparation including the loading time, i. e., transforming the PCC rules, adding constraints for "exactly-one" and the effectivity dates. The column *Check* specifies the time needed for the actual SAT checking and the last column *Total* denotes the total time needed for performing the order checking task.

As can be seen Z3 performs a little better than SMT-RAT on the order checking. This is due to the fact that the preparation takes more time in SMT-RAT as the internal data structures in SMT-RAT might be too complex when only considering propositional logic instead of more expressive logics. Nevertheless SMT-RAT performs in the same order of magnitude than Z3 and therefore is competitive.

The interesting part happens when presorting the orders according to their corresponding product type, i. e., their PCC. Before we needed to load the same PCC multiple times, because their corresponding orders were scattered in the list of all orders. But after sorting the orders every PCC only has to be loaded once and all orders based on this PCC can be checked consecutively. Hence, between orders with the same PCC we only have to pop the previous options and their effectivity dates and push the next options with the new dates. However, we do not have to load the PCC again and save a lot of time. As can be seen, applying the presorting cuts the needed time for checking with SMT-RAT in half, because the preparation time decreases drastically and the presorting only takes 10 milliseconds.

The implementation with Z3 does not benefit from this, because there the PCCs have to be reloaded nevertheless. This is due to the fact that the implementation does not encode the effectivity dates but instead filters the PCC beforehand. That means every rule not effective for the given date is discarded. This approach decreases the amount of rules communicated to the SAT solver, but on the other hand the same PCC must be reloaded for each different date.

Thus, using our approach of encoding the effectivity dates into propositional logic combined with presorting of the orders leads to the best result. Projecting these samples to the order checking

with up to 50,000 examples we might only need approximately 2.5 hours for checking all orders instead of 6 hours with the current implementation.

9.2 PCC checking

Next we evaluate the realizability task which we solved with lemma generation. Here we set the week 49/15 and checked all 368 PCCs for their realizable options.

The results can be seen in Table 9.2.

Future	Solver	Load.	Prep.	Check	Total
✗	SMT-RAT	30.40	54.13	190.70	244.83
	Z3	29.18	112.82	206.11	318.93
✓	SMT-RAT	30.33	53.50	267.25	320.75
	Z3	29.24	142.90	300.58	443.78

Table 9.2: Realizability checking

On the realizability check SMT-RAT performs better than Z3 especially the preparation only takes half the time in comparison to Z3 indicating that our preprocessing of the rules is faster. Further on, the generation of the lemmas decreases the checking time as well. In contrast, the implementation with Z3 uses the naive approach of asserting every option and checking its satisfiability consecutively. Using lemmas also offers the additional possibility to easily get those options which always have to be taken. This computation is not possible with the other approach.

As seen previously with the order checking, the encoding of the effectivity dates gives our approach an advantage when checking for different dates. Therefore the last two rows in the table deal with the realizability check for future dates. Here not only the current date, which we set to 33/15, is considered but also all future dates where some rules change. As we encode the dates already in SMT-RAT this does not change the time for preparation and only the time for checking increases as we have to perform the lemma generation for different dates. In contrast the implementation with Z3 has to reload PCCs for different dates, leading to an increase in the preparation time. As before the checking also takes less time due to the lemmas. In total SMT-RAT only needs 75% of the time of Z3 for the realizability check.

9.3 VarGen checking

Last we consider the VarGen checking which we solved with the use of All-SAT. Here we considered 8 different components each consisting of a set of PCCs and several sets of options to combine. Then each combination of options is checked for each PCC.

The results are shown in Table 9.3. We can see that the total time for each components when using SMT-RAT is nearly half of the time when using Z3. This is primarily due to the fact that the time for checking is mostly one third compared to Z3. Therefore using the approach of All-SAT in contrast to checking each combination individually decreases the time for checking drastically. Especially when considering combinations which are not possible our approach can exclude multiple unsatisfiable combinations at once whereas the other approach has to check them nonetheless.

Components	Solver	Load.	Prep.	Check	Total
Comp. 1	SMT-RAT	31.67	86.55	28.60	115.16
	Z3	32.56	131.12	91.71	222.82
Comp. 2	SMT-RAT	36.16	100.95	32.50	133.45
	Z3	35.67	155.77	96.23	252.00
Comp. 3	SMT-RAT	23.42	60.27	13.26	73.53
	Z3	21.87	65.02	54.77	119.79
Comp. 4	SMT-RAT	6.38	12.95	1.93	14.88
	Z3	5.98	12.78	9.48	22.27
Comp. 5	SMT-RAT	56.47	145.19	34.17	179.36
	Z3	55.36	207.72	146.27	353.99
Comp. 6	SMT-RAT	112.71	335.83	114.72	450.55
	Z3	105.22	502.42	343.01	845.43
Comp. 7	SMT-RAT	33.19	87.33	78.48	165.82
	Z3	29.59	138.95	103.38	242.33
Comp. 8	SMT-RAT	47.04	130.74	38.29	169.03
	Z3	46.38	226.27	126.78	353.05

Table 9.3: VarGen checking

10 Conclusion and future work

10.1 Conclusion

In this thesis we showed the applicability of SAT solving technologies to the product configuration problem in a manufacturing company. As the company already used their own set of rules for constraining the features and options it was fairly straight-forward to encode them into propositional logic. Additionally we believe it will be helpful to directly write new rules in propositional logic as it is more intuitive and also offers a more compact representation, thus avoiding inconsistencies and human errors. As a SAT solver is predestined for deciding whether a given set of rules with additional selected variables is satisfiable, the first task, i. e., the order checking, could be implemented easily and showed the strength of this approach as in contrast to the currently used implementation, we return safe answers and even perform better than the current implementation.

Considering the two remaining tasks we further improved the use of solver technologies by extending our own solver `SMT-RAT` to accommodate for the given problems but at the same time retaining the general purpose character of the solver. The second problem of realizability checking could be solved by generating lemmas capturing the reasons for specific variable assignments. Using lemmas not only helps identifying those variables which never can be satisfied but also those which are always satisfied. Furthermore by giving a reason for each variable assignment this facilitates an engineer to identify inconsistencies and possible problems in the set of rules.

The last task of variance generation could be solved by using All-SAT where all possible solutions are searched. Here we highly benefited from being able to implement All-SAT into `SMT-RAT` instead of triggering it from outside as done with `Z3`. The evaluation shows that our approach drastically improves the performance of the solving process.

Both extensions, lemma generation and All-SAT, now can be found in `SMT-RAT`. Furthermore, we ported `SMT-RAT` to `Windows` and integrated it into the existing implementation of the company enabling the user to choose between using `Z3` or `SMT-RAT`. Thus, our implementation could be used productively in the company.

10.2 Future work

In the future the All-SAT computation could be improved further. If during the solving all clauses are already satisfied but not all variables are assigned, these unassigned variables can take any value. Thus, all extensions of this partial assignment are satisfying assignments meaning that all combinations of unassigned relevant variables are solutions. Hence, we can discard checking them.

Next it might be interesting to develop heuristics for the variable decision in our specific setting. For example it could be beneficial to first choose relevant variables during the solving to guide the solver in first satisfying those. On the other hand this could prove counterproductive as the structure of the rules could favor other variable assignments.

This leads us to the next challenge. As we consider a specific setting here it might be worth extensively analyzing the set of given rules. For example at the moment all rules are implications which could be exploitable. Furthermore we often have the encoding of "exactly-one" which is translated into propositional logic by possible exponential blowup. Implementing our own handling of such an "exactly-one" constraint could drastically improve the solving performance. For example if one of the variables is already satisfied in such a construct, we instantly know that all other variables cannot be satisfied anymore, implying several new variable assignments.

Another interesting point in our setting is the collection of rules in `PCCs`. All our tasks begin by loading a specific `PCC` and usually discarding the previously loaded `PCC`. But those `PCCs` are not always independent of each other. In some cases multiple `PCCs` share similar rules meaning we could only discard parts of the rules and keep other parts which we need in the next `PCC` as well. Identifying those rules offers a huge benefit not only for decreasing the preparation time

but also for the engineers as it allows for new insights about the different PCCs. This could lead to a reduced amount of PCCs which are clearly distinguished from each other. Furthermore the maintenance of the rules would become easier as each change would only need to occur in one rule and must not be carried out through similar rules in different PCCs.

Knowing more about the PCCs also allows to explicitly encode inherent knowledge into the rules. At the moment after a realizability check we gain lemmas stating several variable assignments. But after this check this knowledge is lost. Instead it would be better to permanently store this additional knowledge as it might help improve the solving performance. On the other hand after changing rules these lemmas need to be considered again as they might not be valid anymore and might need to be removed.

In the scope of this thesis we only considered SAT solving techniques. However, in the future it might be useful to also consider more expressive logics as explained in Section 3.3 on page 9. As SMT-RAT is an SMT solver, making this transition is not complicated and we gain the advantage of encoding rules more intuitively and being able to solve other problems. For example the effectivity dates are currently encoded as additional variables which have to be set explicitly in a preprocessing step. Using SMT we can encode these dates as intervals and do not need to apply preprocessing anymore. Furthermore we then could search for specific dates where certain constraints apply. For example we assert a certain option and are now interested in the weeks when this option can be chosen.

Lastly, during our meetings with the company we encountered a lot of other tasks which could be solved using a SMT solver, e. g., scheduling or placing objects in three dimensional space.

Bibliography

[Ake78] S. B. Akers. “*Binary Decision Diagrams*”. *IEEE Trans. Comput.* 27.6 (1978), pages 509–516 (cited on page 1).

[Bar+11] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. “*CVC4*”. *Proc. of CAV.* Volume 6806. LNCS. Springer, 2011, pages 171–177 (cited on page 1).

[Bel+14] A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo. “*SAT COMPETITION 2014*”. 2014. URL: `https://helda.helsinki.fi/handle/10138/135571` (visited on 11/25/2015) (cited on page 1).

[Bie+99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. “*Symbolic Model Checking without BDDs*”. *Proc. of TACAS.* Volume 1579. LNCS. Springer, 1999, pages 193–207 (cited on page 1).

[Bry86] R. E. Bryant. “*Graph-Based Algorithms for Boolean Function Manipulation*”. *IEEE Trans. Comput.* 35.8 (1986), pages 677–691 (cited on page 1).

[BST10] C. Barrett, A. Stump, and C. Tinelli. “*The Satisfiability Modulo Theories Library (SMT-LIB)*”. www.SMT-LIB.org. 2010 (cited on pages 21, 35).

[Cim+13] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. “*The MathSAT5 SMT Solver*”. *Proc. of TACAS.* Volume 7795. LNCS. Springer, 2013, pages 93–107 (cited on page 1).

[Coo71] S. A. Cook. “*The Complexity of Theorem-proving Procedures*”. *Proc. of STOC.* ACM Press, 1971, pages 151–158 (cited on page 1).

[Cor+12] F. Corzilius, U. Loup, S. Junges, and E. Ábrahám. “*SMT-RAT: An SMT-Compliant Nonlinear Real Arithmetic Toolbox (Tool Presentation)*”. *Proc. of SAT.* Volume 7317. LNCS. Springer, 2012, pages 442–448 (cited on page 35).

[Cor+15a] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. “*SMT-RAT - Satisfiability-Modulo-Theories Real Algebra Toolbox*”. 2015. URL: `https://github.com/smtrat/smtrat` (visited on 11/25/2015) (cited on page 35).

[Cor+15b] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. “*SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving*”. *Proc. of SAT.* Volume 9340. LNCS. Springer, 2015, pages 360–368 (cited on pages 1, 35).

[Dan63] G. Danzig. “*Linear programming and extensions*”. 1963 (cited on page 10).

[DDM06a] B. Dutertre and L. De Moura. “*A fast linear-arithmetic solver for DPLL (T)*”. *Proc. of CAV.* Volume 4144. LNCS. Springer. 2006, pages 81–94 (cited on page 10).

[DDM06b] B. Dutertre and L. De Moura. “*Integrating simplex with DPLL(T)*”. Technical report. CSL, SRI International, 2006 (cited on page 10).

[Deh+15] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Ábrahám. “*PROPhESY: A PRObabilistic ParamEter SYnthesis Tool*”. *Proc. of CAV.* Volume 9206. LNCS. Springer, 2015, pages 214–231 (cited on page 1).

[DLL62] M. Davis, G. Logemann, and D. Loveland. “*A Machine Program for Theorem-proving*”. *Communications of the ACM* 5.7 (1962), pages 394–397 (cited on pages 1, 13, 14).

[DMB08] L. De Moura and N. Bjørner. “*Z3: An efficient SMT solver*”. *Proc. of TACAS.* Volume 4963. LNCS. Springer, 2008, pages 337–340 (cited on pages 1, 35).

[DP60] M. Davis and H. Putnam. “*A Computing Procedure for Quantification Theory*”. *Journal of the ACM* 7.3 (1960), pages 201–215 (cited on page 1).

[Dut14] B. Dutertre. “*Yices 2.2*”. *Proc. of CAV.* Volume 8559. LNCS. Springer, 2014, pages 737–744 (cited on page 1).

[ES03] N. Eén and N. Sörensson. “*An Extensible SAT-solver*”. *Proc. of SAT*. Volume 2919. LNCS. Springer, 2003, pages 502–518 (cited on pages 1, 35).

[Fal+11] A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. “*Modeling and Solving Technical Product Configuration Problems*”. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 25.2 (2011), pages 115–129 (cited on page 2).

[Fuh+07] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. “*SAT Solving for Termination Analysis with Polynomial Interpretations*”. *Proc. of SAT*. Volume 4501. LNCS. Springer, 2007, pages 340–354 (cited on page 1).

[GH15] B. Gladman and W. Hart. “*MPIR - Multiple Precision Integers and Rationals*”. 2015. URL: `http://www.mpir.org/` (visited on 11/25/2015) (cited on page 36).

[GN07] E. Goldberg and Y. Novikov. “*BerkMin: A fast and robust Sat-solver*”. *Discrete Applied Mathematics* 155.12 (2007). Proc. of SAT, pages 1549 –1561 (cited on page 1).

[Gra15] T. Granlund. “*GMP - The GNU Multiple Precision Arithmetic Library*”. 2015. URL: `https://gmplib.org/` (visited on 11/25/2015) (cited on page 36).

[GSY04] O. Grumberg, A. Schuster, and A. Yadgar. “*Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis*”. *Proc. of FMCAD*. Volume 3312. LNCS. Springer, 2004, pages 275–289 (cited on page 2).

[KS08] D. Kroening and O. Strichman. “*Decision Procedures: An Algorithmic Point of View*”. 1st edition. Springer, 2008 (cited on page 14).

[KS92] H. Kautz and B. Selman. “*Planning as Satisfiability*”. *Proc. of ECAI*. John Wiley & Sons, Inc., 1992, pages 359–363 (cited on page 1).

[Lee59] C. Y. Lee. “*Representation of Switching Circuits by Binary-Decision Programs*”. *Bell System Technical Journal* 38.4 (1959), pages 985–999 (cited on page 1).

[Lev73] L. A. Levin. “*Universal sequential search problems*”. *Problemy Peredachi Informatsii* 9.3 (1973), pages 115–116 (cited on page 1).

[MA04] E. R. van der Meer and H. R. Andersen. “*BDD-based recursive and conditional modular interactive product configuration*”. *Proc. CSPIA*. 2004, pages 112–126 (cited on page 2).

[Mos+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. “*Chaff: Engineering an Efficient SAT Solver*”. Proc. of DAC. ACM Press, 2001, pages 530–535 (cited on pages 1, 17).

[MS99] J. a. P. Marques-Silva. “*The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*”. *Proc. of EPIA*. Volume 1695. LNCS. Springer, 1999, pages 62–74 (cited on page 17).

[MSS96] J. a. P. Marques-Silva and K. A. Sakallah. “*GRASP - A New Search Algorithm for Satisfiability*”. Proc. of ICCAD. IEEE Computer Society, 1996, pages 220–227 (cited on page 1).

[NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. “*Abstract DPLL and Abstract DPLL Modulo Theories*”. *Proc. of LPAR*. Volume 3452. LNCS. Springer, 2005, pages 36–50 (cited on page 9).

[Rob65] J. A. Robinson. “*A Machine-Oriented Logic Based on the Resolution Principle*”. *Journal of the ACM* 12.1 (1965), pages 23–41 (cited on pages 1, 14).

[SKK00] C. Sinz, A. Kaiser, and W. Küchlin. “*SAT-Based Consistency Checking of Automotive Electronic Product Data*”. *Proc. of ECAI*. IOS Press, 2000, pages 74–78 (cited on page 2).

[SKK01] C. Sinz, A. Kaiser, and W. Küchlin. “*Detection of Inconsistencies in Complex Product Configuration Data Using Extended Propositional SAT-Checking*”. *Proc. of FLAIRS*. AAAI Press, 2001, pages 645–649 (cited on page 2).

[SLM92] B. Selman, H. Levesque, and D. Mitchell. “*A New Method for Solving Hard Satisfiability Problems*”. *Proc. of AAAI*. AAAI Press, 1992, pages 440–446 (cited on page 1).

[Tse68] G. S. Tseitin. “*On the complexity of proof in prepositional calculus*”. *Zapiski Nauchnykh Seminarov POMI* 8 (1968), pages 234–259 (cited on page 4).

[Tuc+07] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. "*OPIUM: Optimal Package Install/Uninstall Manager*". *Proc. of ICSE.* IEEE Computer Society, 2007, pages 178–188 (cited on page 1).

[Yu+14] Y. Yu, P. Subramanyan, N. Tsiskaridze, and S. Malik. "*All-SAT Using Minimal Blocking Clauses*". *Proc. of VLSI Design.* IEEE Computer Society. 2014, pages 86–91 (cited on page 2).

[ZM03] L. Zhang and S. Malik. "*Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications*". *Proc. of DATE.* IEEE Computer Society. 2003, pages 10880–10885 (cited on page 27).

[Mic15] Microsoft Research. "*Z3 Theorem Prover*". 2015. URL: `https://github.com/Z3Prover/z3` (visited on 11/25/2015) (cited on page 35).