Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

SOLVING PSEUDO-BOOLEAN CONSTRAINTS

Marta Grobelna

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

Additional Advisor:
Gereon Kremer, M.Sc.

Aachen, July 3, 2017

Abstract

Boolean Satisfiability (SAT) became more and more important in the recent years. It offers an approach for solving many practical problems in various areas of application, such as *Electronic Design Automation* (EDA), in a very efficient way. Many practical problems can be compactly represented by *pseudo-Boolean (PB) constraints*. Therefore, in the last few years PB-solvers have been developed. A common way to solve PB-constraints is to encode each PB-constraint as a satisfiability-equivalent Boolean formula and use a SAT-solver in order to decide the satisfiability. However, in recent years many SMT-solvers were developed offering efficient approaches which could be used for solving PB-constraints. This thesis proposes an SMT-solver that solves pseudo-Boolean satisfiability problems. In doing so the PB-constraints are encoded as Boolean and arithmetic formulas and combined into a liner integer arithmetic formula. Moreover, two approaches for simplifying PB-constraints are presented which may accelerate the solving procedure.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Marta Grobelna
Aachen, den 3. Juli 2017

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

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.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

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:

Ort, Datum

Unterschrift

Contents

Chapter 1

Introduction

Boolean Satisfiability became a powerful technology during the past few decades. This trend persists due to the remarkable number of applications for satisfiability (SAT) solver in, e.g., model checking, automated planning and scheduling, combinatorial design or circuit design verification. SAT-solvers are even used in software engineering. Since the complexity of software has dramatically increased in recent years, the complexity of corresponding *Unified Modeling Language (UML)* models, which play a key role in today's software engineering, has also increased. As the quality of software depends on the quality of its UML model, it is crucial to verify the models. This can be done by a SAT-solver as presented in [SWK+10]. Another interesting application field is the *Electronic Design Automation (EDA)* in particular, automatic test pattern generation, circuit delay computation, crosstalk noise analysis and so forth [BBH+09, CESS08, CK05].

One fundamental question arises at this point - how must a given problem be represented such that a SAT-solver can be used to solve it? One way is to represent the problem using *propositional logic*. In this case the SAT-solver gets a formula in, e.g., *Conjunctive Normal Form (CNF)* as input and decides if the given formula is satisfiable. Unfortunately, propositional logic is not expressive enough to represent many problems. This is why the more powerful *first-order logic* is commonly used. However, one cannot use a simple SAT-solver as decision procedure for it. Proving the satisfiability of a *general* first-order logic formula requires a *first-order theorem prover*. The semantics of function, predicate and constant symbols in a general first-order formula are arbitrary. Obviously, it is not sensible to decide the satisfiability of an interpretation that is not sensible for the proper application. This is why in practice the formulas representing a problem usually refer to a certain *theory* which fixes the meaning of the function, predicate and constant symbols [BBH+09]. A popular theory is the *Presburger arithmetic*. It contains the symbols $(0, 1, +, -, \leq)$ in which 0 and 1 are constant symbols, + and - are binary function symbols and $\leq$ is a binary predicate symbol. All symbols have their usual mathematical meaning. Another prominent theory is the theory of *integer linear arithmetic with equality*. It contains the addition and multiplication functions, as well as the following binary predicates: $<$, $\leq$, $=$, $\geq$, $>$. Solver which are able to decide the satisfiability of first-order formulas referring to a certain underlying theory are called *SAT Modulo Theories (SMT)* solvers [BBH+09, Kra95].

SMT-solvers usually use a SAT-solver as an underlying decision procedure. The

question is how to use a SAT-solver for solving the SMT-problem. Basically, there are two kinds of SMT-solvers: *eager SMT-solvers* and *lazy SMT-solvers*. An eager SMT-solver first translates the original formula into a satisfiability-equivalent propositional formula and afterwards the SAT-solver has to decide upon its satisfiability. A lazy SMT-solver abstracts from the theory atoms and uses a SAT-solver to generate solutions for the Boolean structure of the formula. For each solution, a theory solver is used to decide whether the respective constraints are consistent. A prominent theory solver for linear arithmetic theory over $\mathbb{Q}$ is *Simplex*. However, it depends on the theory which kind of solver should be applied, as the translation to satisfiability-equivalent propositional formula cannot be made as efficiently for all kinds of theories.

The focus of this thesis lies on the *pseudo-Boolean theory*. A pseudo-Boolean constraint can be written as a polynomial which variables can only take the values *0* or *1* while the coefficients are arbitrary integers. A special case of the PB-constraints are the *cardinality* constraints. Those are constraints which variables can also only take the values *0* or *1* but all coefficients equal to 1. This kind of constraints is used for, e.g., binate covering based technology mapping, constraint-based placement and routing, noise analysis and other [BBH+09, CK03, Seb07].

There are problems which can be expressed by a formula in propositional logic or in first-order logic or in both of them. However, not every representation is optimal. It has turned out that the aforementioned EDA problems can be described by a set of *pseudo-Boolean (PB)* constraints more compactly than by propositional logic [CK03]. This also holds for many other problems, e.g., VLSI design, statistical mechanics, maximum satisfiability, economics, manufacturing and so forth. Therefore, it is sensible to develop PB-solvers [BBH+09, BH02].

Most practical questions are either *satisfiability* or *optimization* problems. Satisfiability problems refer to the following category of problems: given a PB-constraint, decide if the formula is satisfiable. If it is satisfiable then give an assignment of variables which satisfies the formula. Otherwise, an infeasible subset of constraints should be returned. Optimization problems consist of an objective function which has to be minimized or maximized subject to a set of PB-constraints [BBH+09].

This thesis proposes an SMT-solver which solves pseudo-Boolean satisfiability problems. The solver consists of a SAT-solver and a Simplex-solver. Before the SMT-solver is applicable, a preprocessor encodes the PB-constraints as linear integer arithmetic formulas where one part of the formulas are Boolean formulas and the other part are arithmetic formulas. Moreover, two methods for simplifying PB-constraints are presented. One approach uses *residual number bases* [FC14] in order to reduce the number of occurring variables in a constraint. The other approach implements the well-known *Gauss algorithm* that reduces the number of constraints.

This thesis is structured as follows. In chapter 2 pseudo-Boolean and cardinality constraints are defined. Moreover, it gives an theoretical background about the translation of PB-constraints. Then in chapter 3 the encoding of PB-constraints is discussed. The first part of chapter 3, deals with the decision which PB-constraints should be encoded as Boolean formulas and how they can be encoded. The second part deals with the encoding of arithmetic formulas. In chapter 4, both simplifying approaches are presented. Then in chapter 5 the results of benchmarks are presented. Finally, in chapter 6 the thesis concludes with a short summary and future prospect.

Chapter 2

Preliminaries

In this chapter all necessary definitions are presented. In the first section, linear and non-linear pseudo-Boolean constraints are defined. The next section deals with a special case of pseudo-Boolean constraints - the cardinality constraints. Finally, in the last section some terms needed for the encoding are explained.

2.1 Pseudo-Boolean Constraints

A *pseudo-Boolean function* f is an n-ary function with $f : \mathbb{B}^n \mapsto \mathbb{Z}$, where $\mathbb{B}$ is the set of Boolean values *0* and *1*, and $\mathbb{Z}$ is the set of integers. A *pseudo-Boolean constraint* (PB-constraint) is an equation or an inequality between a pseudo-Boolean function and an integer. Moreover, it can be either linear or non-linear. A non-linear PB-constraint has the form

$$\sum_i a_i \cdot \prod_j l_{ij} \ \# \ b$$

where the coefficients a_i and the right hand side b are integers, $\#$ is one of the relations $<$, $\leq$, $=$, $\neq$, $\geq$, $>$, and l_{ij} are the *literals*. A literal l_{ij} can either be a Boolean variable x_{ij} or its negation $\overline{x}_{ij}$. For linear PB-constraints, as the name already suggests, the multiplication of literals is not allowed. Hence, linear PB-constraints have the form

$$\sum_i a_i \cdot l_i \ \# \ b$$

All predicate and function symbols have their usual mathematical meaning. As the variables are Boolean, a variable assigned to *true* is interpreted as 1 while a variable assigned to *false* is interpreted as 0. Thus, all predicate and function symbols can be used as usual. A *solution* of a PB-constraint is an assignment of variables which satisfies the constraint. If a constraint is satisfied by every possible assignment, then the constraint can be simplified to *true* and will be called *tautology*. If there exists no assignment such that the constraint is satisfied, then the constraint can be simplified to *false* and will be called *contradiction* [Bar96, BBH+09].

2.2 Cardinality Constraints

A special case of PB-constraints are the *cardinality constraints*. There are three different types of cardinality constraints: $atleast(b, \{l_1, \ldots, l_n\})$, $atmost(b, \{l_1, \ldots, l_n\})$ and $exactly(b, \{l_1, \ldots, l_n\})$. The constraint $atleast(b, \{l_1, \ldots, l_n\})$ requires that at least b literals among $l_1, \ldots, l_n$ are assigned to *1*. The constraint $atmost(b, \{l_1, \ldots, l_n\})$ requires that at most b literals among $l_1, \ldots, l_n$ are assigned to *1*. Finally, the constraint $exactly(b, \{l_1, \ldots, l_n\})$ requires that exactly b literals among $l_1, \ldots, l_n$ are assigned to *1*. All three cardinality constraints can be represented by a PB-constraint as follows

$$atleast(b, \{l_1, \ldots, l_n\}) \iff \sum_{i=1}^{n} 1 \cdot l_i \geq b$$

$$atmost(b, \{l_1, \ldots, l_n\}) \iff \sum_{i=1}^{n} 1 \cdot l_i \leq b$$

$$exactly(b, \{l_1, \ldots, l_n\}) \iff \sum_{i=1}^{n} 1 \cdot l_i = b$$

Note that all coefficients in a PB-constraint representing a cardinality constraint are equal to 1. However, if all coefficients are equal to -1 the constraint can easily be transformed into an equivalent cardinality constraint. For example, the constraint $-1x_1 - 1x_2 - 1x_3 - 1x_4 \geq -1$ can be transformed to $1x_1 + 1x_2 + 1x_3 + 1x_4 \leq 1$. Hence, it is equivalent to an *atmost* cardinality constraint. Moreover, if all coefficients on the left hand side and the integer on the right hand side are equal, then the constraint can be transformed into an equivalent cardinality constraint by dividing the constraint by the integer. For example, the constraint $2x_1 + 2x_2 + 2x_3 + 2x_4 \leq 2$ is equivalent to $x_1 + x_2 + x_3 + x_4 \leq 1$. Thus, the constraint is equivalent to an *atmost* cardinality constraint [BBH+09].

2.3 Translating Pseudo-Boolean Constraints

All PB-constraints can be encoded as propositional formulas. In doing so, one needs to find a propositional formula which is only *satisfiability-equivalent* to the PB-constraint. This means, that all assignments $\mathfrak{I}$ of the variables occurring in the PB-constraint φ that satisfy φ ($\mathfrak{I} \vDash \varphi$), must also satisfy the corresponding propositional formula ψ ($\mathfrak{I} \vDash \psi$). Moreover, there must not exist a different assignment $\mathfrak{I}'$ of the literals occurring in φ, such that $\mathfrak{I}' \nvDash \varphi$ but $\mathfrak{I}' \vDash \psi$ and vice verse. This can be also written as follows

Definition 2.3.1. *Assume two formulas φ and ψ and there exists an assignment $\mathfrak{I}$ with $\mathfrak{I} \vDash \varphi$. Then the two formulas are satisfiability-equivalent if and only if there exists an assignment $\mathfrak{I}'$ with $\mathfrak{I}' \vDash \psi$.*

However, encoding PB-constraints as propositional formulas can lead to formulas with exponential number of constraints [BBH+09]. Especially for very long PB-constraints the according propositional formula might be very complicated. Even more complicated is the translation for PB-constraints that have coefficients with different signs. Therefore, some of the PB-constraints are usually encoded as integer arithmetic formulas [BA12, BBH+09].

Similarly as for the propositional formulas, all PB-constraints can also be encoded as integer arithmetic formulas. Here, one has to pay attention to the fact that the

domain of the literals of an integer arithmetic formula is no more restricted to Boolean values. Thus, each literal can take any integer value. Obviously, this is a problem that must be considered when encoding PB-constraints as integer arithmetic formulas. The exact procedure is presented in the next chapter.

Chapter 3

Encoding Pseudo-Boolean Constraints

3.1 General Procedure

This thesis proposes an SMT-solver for linear integer arithmetic which is used for solving linear PB-constraints. Due to the fact that the SMT-solver cannot directly process PB-constraints, a preprocessor is needed which encodes the PB-constraints as Boolean and arithmetic formulas and combines them into one liner integer arithmetic formula. This section gives a short overview over the functioning of the preprocessor and the SMT-solver. In the next two sections more detail information about the translation follow.

The general approach is modeled by the flowchart 3.1. The blue elements represent processes which are done by the preprocessor. The input for the preprocessor is a conjunction over PB-constraints which is represented by a list of the PB-constraints. The preprocessor takes a PB-constraint from the list and checks if it can be encoded as a propositional formula. Otherwise, the PB-constraint is encoded as an integer arithmetic formula. Afterwards, the preprocessor again checks if there is an another PB-constraint in the list and repeats the procedure. Once all constraints are encoded, the constraints are combined into one liner integer arithmetic formula. The satisfiability of the formula is then decided by the SMT-solver that consists of a SAT and Simplex solver.

3.2 Encoding as Propositional Formula

The focus of this section lies on the decision if a PB-constraint should be translated into a satisfiability-equivalent propositional formula and how it can be done. There are many PB-constraints which can easily be encoded as propositional formulas, as they directly correspond to a Boolean construct, e.g., implication. For example, the PB-constraint $-1x_1 + 1x_2 \geq 1$ can be encoded as $x_1 \rightarrow x_2$. There are many such PB-constraints and those can be processed by a SAT-solver very efficient [BBH+09].

One of the tasks of the preprocessor is to filter out constraints which should be encoded as propositional formulas. It should be possible to create the corresponding propositional formula fast and, even more important, the SAT-solver should be able

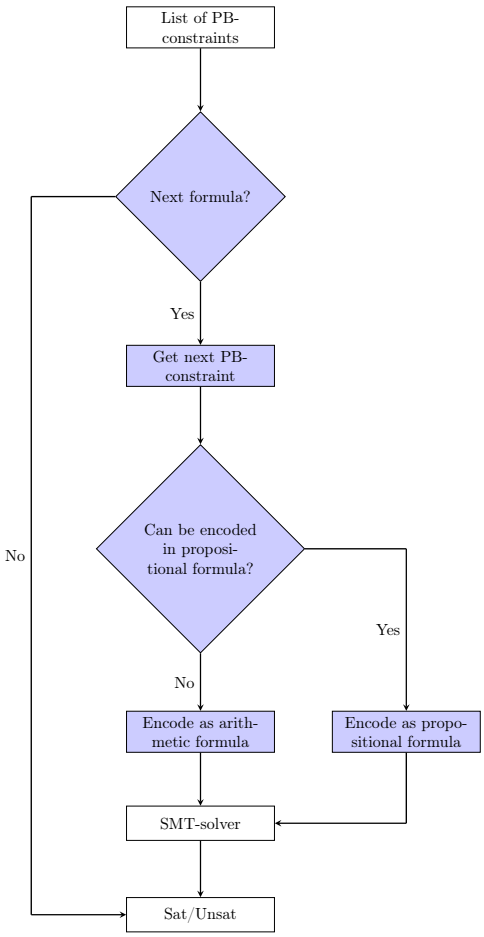

Figure 3.1: Approach of the preprocessor.

to process this formula faster than the Simplex-solver. The most intuitive way to filter out such constraints is to look at the number of terms on the left hand side of the constraint. It is sensible to translate all PB-constraints which have only one term on the left hand side, as all of them directly correspond to an easy Boolean construct. This criterion divides the PB-constraints into two categories: short and long formulas. Short formulas are formulas which have only one term on the left hand side. Respectively, long formulas consist of more than one term. While all short formulas can simply be encoded as propositional formulas, it is quite hard to decide if it is sensible to encode a long formula. Thus, one needs further criteria, e.g., the consistency of the coefficient's signs. Depending on the right hand side, some long PB-constraints with consistent signs can be translated efficiently. For others, some more special criteria are required.

Due to the definition 2.3.1, finding a correct encoding for a PB-constraint requires the consideration of all possible assignments of literals occurring in the constraint. All assignments which satisfy the considered PB-constraint, must also satisfy the corresponding propositional formula and no other assignment must satisfy it.

The first subsection deals with the encoding of short formulas. The next subsection describes the translation of long formulas with consistent signs. Afterwards, subsection 3.2.3 presents the encoding of cardinality constraints. Finally, in subsection 3.2.4 the encoding of long formulas with non-consistent signs is explained.

3.2.1 Encoding Short Formulas

This section deals with the short formulas, i.e., formulas that have the form $a \cdot x \ \# \ b$. The case-by-case analysis is structured as follows. First the encoding of PB-constraints with the relations $\geq$ and $>$ are explained. The case-by-case analysis for those PB-constraints is divided according to the sing of a. The next considered group of PB-constraints are the equations. The last group of PB-constraints are those with the relation $\neq$.

The first case-by-case analysis considers PB-constraints with the relations $\geq$ and $>$, and assumes that a is positive.

$$ax_1 \overset{>}{\geq} b \rightsquigarrow \begin{cases} true, & b < 0 \\ true, & b = 0 \text{ and the relation is } \geq \\ x_1, & b > 0 \text{ and } a > b \\ x_1, & b = 0 \text{ and the relation is } > \\ x_1, & a = b \text{ and the relation is } \geq \\ false, & a = b \text{ and the relation is } > \\ false, & b > a \end{cases} \tag{3.1}$$

In the first case the PB-constraint can be encoded as *true*, as a positive number is always strictly greater than a negative one. In the second case the PB-constraint can also be encoded as *true* due to the fact that b is equal to 0 and the minimal value reachable on the left hand side is also equal to 0. In the next case b is positive but it is less or equal to a. Thus, the case when the left hand side is equal to 0 must be excluded. This is done by encoding the formula as x_1. In the next case the formula can also be encoded as x_1, since only a positive number can be strictly greater than 0. The next two cases correspond to short PB-constraints where the coefficient and the right hand side are equal. Obviously, when the relation is $\geq$ the literal must be set to *1* is order to satisfy the formula. For the relation $>$ the formula can never be satisfied and can be encoded as *false*. Finally, in the last case b is greater than a. Hence, such PB-constraint can obviously be never satisfied.

The previous proof by cases considered PB-constraints with $a > 0$. Now, the cases where a is negative are discussed.

$$ax_1 \overset{>}{\geq} b \rightsquigarrow \begin{cases} false, & b > 0 \\ false, & b = 0 \text{ and the relation is } > \\ \overline{x}_1, & b < 0 \text{ and } b > a \\ \overline{x}_1, & b = 0 \text{ and the relation is } \geq \\ \overline{x}_1, & a = b \text{ and the relation is } > \\ true, & a = b \text{ and the relation is } \geq \\ true, & b < a \end{cases} \tag{3.2}$$

Due to the fact that a is negative and in the first case b is positive, it is trivial that the PB-constraint has to be encoded as *false*. The second case is also trivial, as negative

number is not greater than 0. In the third case b is also negative, however it is greater than a. Thus, the formula will only be satisfied if the left hand side is equal to 0. Therefore, such PB-constraints has to be encoded as the negation of the literal. In the next case b is again equal to 0. Hence, the left hand side must also be equal to 0. The next two cases consider PB-constraints where the coefficient is equal to the right hand side. If the relation is $>$, the left hand side must be equal 0 since the right hand side is negative. Otherwise, if the relation is $\geq$ then the formula is always satisfied. The last case is trivial.

That was the case-by-case analysis for PB-constraints with the relations $\geq$ and $>$. The next step would be to look at PB-constraints which have the relations $\leq$ and $<$. However, this cases do not have to be considered because they can be transformed to the cases presented previously, as it hols that

$$\sum_i a_i x_i \overset{<}{\leq} b \iff \sum_i -a_i x_i \overset{>}{\geq} -b \tag{3.3}$$

This is why the case-by-case analysis for those PB-constraints is not necessary. Instead of this, the next case-by-case analysis considers the equations.

$$ax_1 \;=\; b \rightsquigarrow \begin{cases} x_1, & a = b \\ \overline{x}_1, & b = 0 \\ false, & a \neq b \end{cases} \tag{3.4}$$

The first case is trivial since when a and b are equal, the constraint can only be satisfied if x_1 is set to *1*. The situation is different when b is equal to zero, the constraint is satisfied if x_1 is assigned to *0*. This is the reason why, such constraints are encoded as the negation of the literal. The last case is trivial.

The last case-by-case analysis considers PB-constraints with the relation $\neq$.

$$ax_1 \;\neq\; b \rightsquigarrow \begin{cases} x_1, & b = 0,\ a \neq 0 \\ \overline{x}_1, & b = a \\ true, & a \neq b \end{cases} \tag{3.5}$$

In the first case the constraint has to be translated to x_1 because a must not be equal to zero. In the next case b and a are equal. Thus, the only possibility to make the left and right side unequal is to make the left hand side equal to 0. The last case is again trivial.

Those were all cases for short formulas. In the next sub-section the translation of more complex PB-constraints is discussed.

3.2.2 Encoding Long Formulas with Consistent Signs

The PB-constraints considered in this section have an arbitrary number of terms and all coefficients on the left hand side are either positive or negative. A new criterion used for filtering out the constraints of interest is the sum over all coefficients on the left hand side. Nevertheless, a PB-constraint with positive (negative) left hand side is a constraint where all coefficients on the left hand side are positive (negative).

The case-by-case analysis in this section is structured as follows. First, PB-constraints with the relations $\geq$ and $>$ are considered. The case-by-case analysis for them is divided according to the sings of the coefficients on the left hand side.

Thus, the first part of the case-by-case analysis considers PB-constraints with positive left hand side. Then those with negative left hand side are considered. Afterwards equations and PB-constraints with the relation $\neq$ are considered.

The first case-by-case analysis refers to PB-constraints with the relations $\geq$ and $>$ and positive left hand side. Now, there are only two criteria left - the value of b and the relations. First, the cases where b is equal to 0 are considered.

$$\sum_i a_i \cdot x_i \overset{>}{\geq} 0 \rightsquigarrow \begin{cases} true, & \text{the relation is } \geq \\ \bigvee_i x_i, & \text{the relation is } > \end{cases} \tag{3.6}$$

Due to the fact that the left hand side is positive the constraint is valid, independent from the assignment of the literals x_i. The highest value which can be reached on the left hand side is equal to the sum over all coefficients ($x_i = 1$ for all i). The minimum value is equal to zero ($x_i = 0$ for all i). This is the reason why in the first case the constraint can be encoded as *true*. In the next case one has to pay attention to the fact that the left hand side is not allowed to be equal to 0. This means that the case where all x_i are assigned to *0* has to be excluded. Since a disjunction is true if and only if at least one literal is true, and false if and only if all literals are false, the constraint can be encoded as a disjunction over all literals. Now it is ensured that the minimum value which can be reached on the left hand side is equal to the smallest coefficient. Since the smallest coefficient is greater than 0 (all coefficients are positive), the translation is correct.

The next considered case assumes that b is negative. Since the left hand side is positive and the relation is either $\geq$ or $>$, such a constraint is a tautology. Hence, the constraint is encoded as *true*.

The last case considers the situation where b is positive. This is a quite complicated situation but first the clear cases are discussed.

$$\sum_i a_i \cdot x_i \overset{>}{\geq} b \rightsquigarrow \begin{cases} false, & sum < b \\ \bigwedge_i x_i, & sum = b \text{ and the relation is } \geq \\ false, & sum = b \text{ and the relation is } > \end{cases} \tag{3.7}$$

In this case one has to consider the sum of the constraints on the left hand side. The sum can be seen as the maximum value which can be reached on the left hand side ($x_i = 1$ for all i). Therefore, if already the maximum reachable value on the left hand side is smaller than b, it is not possible to satisfy the constraint. Therefore, in the first case the constraint is encoded as *false*. In the next case the sum over all coefficients is equal to b. This corresponds to a conjunction over all variables, since a conjunction is true if and only if all literals are true. The situation is different when it is required that the left hand side is strictly greater than b. As the sum is equal to b there exists no possibility to make the left hand side greater. Hence, the constraint is unsatisfiable. Observe that the case where the sum is strictly greater than b is not considered. The reason why this case is not considered is that one has to enumerate all possibilities which satisfy this condition. The following example illustrates the problem. Consider the following inequality

$$2 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 + 1 \cdot x_4 + 1 \cdot x_5 > 3 \tag{3.8}$$

It is fulfilled for many different assignments of the literals x_i. Finding all models for this inequality is similar to the *subset sum problem*. This is a well-know NP-complete

problem. The input for this problem is a set of positive integers and the question is whether there exists a subset of the integers such that the sum over these is equal to an integer S [Woe03]. Here, the problem is similar. A set of positive integers is given. The task is to find all subsets whose sum is strictly greater than the right hand side of the inequality. Depending on the number of terms, as well as the integer on the right hand side, the number of such subsets can be very high. Thus, for such PB-constraints the arithmetic encoding is usually more suitable.

In the previous case-by-case analysis the situation where the left hand side is positive has been considered. Now, the cases where the left hand side is negative are discussed. First, the case where b is equal to 0 is considered.

$$\sum_i a_i \cdot x_i \overset{>}{\geq} 0 \rightsquigarrow \begin{cases} false, & \text{the relation is } > \\ \bigwedge_i \overline{x}_i, & \text{the relation is } \geq \end{cases} \tag{3.9}$$

In the first case the formula can be encoded as *false*, since all coefficients are negative (and so the sum over them) and a negative number is not greater than 0. Since the coefficients are negative, the sum over them can be interpreted as the smallest reachable value on the left hand side. Consequently, the highest value will be reached if all literals x_i are assigned to *0*. Thus, in the second case all literals have to be assigned to *0* because for all the other assignments the sum would be negative. Therefore, in this case the constraint is encoded as a conjunction over the negated literals x_i.

The next considered case is where b is greater than 0. Due to the fact that the sum over the coefficients is negative neither a constraint with the relation $\geq$, nor a constraint with the relation $>$ can be satisfied when the coefficients are negative and b is positive. Therefore, in both cases the constraint is simply encoded as *false*.

The last possibility for b is left. Here, the cases where b is negative are discussed.

$$\sum_i a_i \cdot x_i \overset{>}{\geq} b \rightsquigarrow \begin{cases} true, & sum > b \\ true, & sum = b \text{ and the relation is } \geq \\ \bigvee_i \overline{x}_i, & sum = b \text{ and the relation is } > \end{cases} \tag{3.10}$$

Clearly, if the sum over the coefficients is strictly greater than b, no assignment can make the left hand side smaller than b. Therefore, such PB-constraints are encoded as *true*. In the next case, the PB-constraint can also be encoded as *true*, due to the fact that no assignment can make the left hand side less than b. The last case is also quite clear. It must be avoided that all literals are assigned to *1*. For all the other cases, the left hand side is strictly greater than b. Therefore, a disjunction over the negated literals is a correct encoding for such PB-constraints. However, there again exists one more case, namely when the sum is strictly less than b. This case is also similar to the subset sum problem. Hence, arithmetic encoding for this cases is usually more suitable.

Due to the same reasons as previously, it is unnecessary to pay attention to cases where the relation is $\leq$ or $<$, since they can also be converted to the cases presented above. The next case-by-case analysis considers equations.

$$\sum_i a_i \cdot x_i = b \rightsquigarrow \begin{cases} \bigwedge_i x_i, & sum = b \\ \bigwedge_i \overline{x}_i, & sum \neq b \text{ and } b = 0 \end{cases} \tag{3.11}$$

In the first case the sum over coefficients is equal to b. This implies that all literals x_i must be set to *1*. Therefore, the constraint is encoded as a conjunction over the literals. In the second case b is equal to zero. In this situation the left hand side must be equal to 0 in order to fulfill the constraint. Thus, all literals must be set to *0* and therefore the constraint is encoded as a conjunction over negated literals. If b is not equal to zero and the sum is not equal to b, then the arithmetic encoding is usually more efficient than the Boolean encoding.

The last case-by-case analysis considers PB-constraints with the relation $\neq$. In this case there exists only one situation when it pays off to translate the constraint into propositional formula. If the right hand side of the constraint is equal to 0, the constraint can be translated into the formula $\bigvee_i \overline{x}_i$. This translation is valid because this formula is equivalent to $\neg(\bigwedge_i x_i)$ and since the left hand side should not be equal to the right hand side, at least one literals has to be set to *1*. For all the other cases the arithmetic encoding may be better than Boolean due to reasons explained before.

3.2.3 Encoding Cardinality Constraints

The focus of this section lies on the translation of cardinality constraints. These are constraints whose coefficients are equal to 1 and therefore they are special cases of usual PB-constraints. Due to their quite easy structure, it is possible to translate some of them efficiently to propositional formulas. This section only considers cardinality constraints which frequently occur in benchmarks from the *Pseudo-Boolean Competition* in the years 2016 and 2015. In order to simplify the case-by-case analysis first, the translation of the general three kinds of cardinality constraints are discussed.

First, the encoding of the constraint $exactly(b,\{x_1,\ldots,x_n\})$ is explained. It corresponds to the following PB-constraint

$$x_1 + x_2 + \ldots + x_n = b$$

This constraint is satisfied if exactly b literals on the left hand side are assigned to *1* and can be encoded as the following propositional formula

$$\bigvee_{i_1=1}^{n} \bigvee_{\substack{i_2=1 \\ i_2 \neq i_1}}^{n} \cdots \bigvee_{\substack{i_b=1 \\ i_b=i_{b-1}}}^{n} \bigwedge_{\substack{i_{b+1}=1 \\ i_{b+1}\neq i_b}}^{n} (x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_b} \wedge \overline{x}_{i_{b+1}} \wedge \ldots \wedge \overline{x}_{i_n}) \tag{3.12}$$

The first b literals must be assigned to *1*, the remaining n - b must be negated, in order to satisfy the formula.

Now, the $atleast(\mathrm{b}, \{x_1,\ldots,x_n\})$ constraint is considered. Every *atleast* constraint can be written as the following PB-constraint

$$x_1 + x_2 + \ldots + x_n \geq b$$

It requires that at least b literals on the left hand side are assigned to *1*, which in turn means that at most n - b literals have to be assigned to *0*. Trivially, the formula cannot be satisfied if the number of literals n is less than b. Therefore, here $n \geq b$ is assumed. In order to encode this formula, one can use the encoding of the constraint *exactly* to exclude the cases where less than b literals are assigned to *1*. Hence, the constraint can be translated as follows

$$\begin{aligned} &\bigvee_{i=1}^{n} x_i \wedge \neg\, exactly(b-1,\{x_1,\ldots,x_n\}) \wedge \\ &\neg\, exactly(b-2,\{x_1,\ldots,x_n\}) \wedge \ldots \wedge \neg\, exactly(1,\{x_1,\ldots,x_n\}) \end{aligned} \tag{3.13}$$

The first sub-formula guarantee that at least one literal is assigned to *1*. However, if b is greater than 1, one must exclude the cases where less than b literals are assigned to *1*. Here, the encoding for *exactly(k,*$\{x_1, \ldots, x_n\}$*)* can be used, as one has to ensure that not only b - 1, b - 2, b - 3, and so on, literals are set to *1*.

The cardinality constraint *atmost*(b, $\{x_1, \ldots, x_n\}$) remains. This will be unsatisfiable if b is a negative number. Thus, it is assumed that b is positive. This constraint is equivalent to the following PB-constraint

$$x_1 + x_2 + \ldots + x_n \leq b \tag{3.14}$$

This formula will be satisfied if at most b literals on the left hand side are assigned to *1*. This leads to a propositional formula which is similar to the previous one. The difference between those two is that here the number of literals which are not negated is increased and not, as before, the number of negated literals. Hence, the formula has the form

$$\begin{aligned} &\mathit{exactly}(0, \{x_1, \ldots, x_n\}) \ \vee \ \mathit{exactly}(1, \{x_1, \ldots, x_n\}) \\ &\qquad \vee \ldots \vee \ \mathit{exactly}(b, \{x_1, \ldots, x_n\}) \end{aligned} \tag{3.15}$$

The first sub-formula ensures that no literal can be assigned to *1*. Together with the next sub-formula it is allowed to assign at most one literal to *1*. The next one allows assigning two literals to *1* and so on.

Now, since the general encoding of the three kinds of cardinality constraints are clear, a closer look at the constraints which frequently occur in benchmarks can be taken. Usually, the constraints occurring in benchmarks are not directly in the form of one of those three kinds of cardinality constraints. However, they can be transformed to one of them. The first case-by-case analysis considers PB-constraints with the relation $\geq$. The PB-constraint

$$x_1 + x_2 + \ldots + x_n \geq 1$$

occurs very oft and it can be encoded very efficiently. Obviously, this constraint is equivalent to the cardinality constraint *atleast*(1, $\{x_1, \ldots, x_n\}$). Therefore, it can be encoded just as a disjunction over the literals, as explained in 3.13. Another interesting constraint is the following

$$-x_1 - x_2 - \ldots - x_n \geq -1 \iff x_1 + x_2 + \ldots + x_n \leq 1$$

This PB-constraint is not a cardinality constraint. Thus, first the both sides of the constraint must be multiplied by -1. This is clearly equivalent the cardinality constraint *atmost*(1, $\{x_1, \ldots, x_n\}$). The last constraint belonging to this category is the PB-constraint

$$-x_1 - x_2 - \ldots - x_n \geq -2 \iff x_1 + x_2 + \ldots + x_n \leq 2 \tag{3.16}$$

The difference between this constrain and the previous one is the right hand side. Therefore, the constraint is encoded as *atmost*(2, $\{x_1, \ldots, x_n\}$).

The remaining type of PB-constraints are those which are equivalent to the cardinality constraint *exactly*(b, $\{x_1, \ldots, x_n\}$). In benchmarks from the *Pseudo-Boolean Competition* in the years 2016 and 2015, the most frequently occurring cardinality constraints are *exactly*(0, $\{x_1, \ldots, x_n\}$) and *exactly*(1, $\{x_1, \ldots, x_n\}$). Thus, only those are encoded as propositional formulas.

3.2.4 Encoding Long Formulas with Non-Consistent Signs

PB-constraints with non-consistent sings are those that have negative and positive coefficients on the left hand side. Depending on the sum over the coefficients and the right hand side, the Boolean encoding is usually complicated, as those are also similar to the subset sum problem mentioned before. Therefore, most of PB-constraints belonging to this category are encoded as arithmetic formulas. However, there are few of them that directly correspond to quite easy Boolean constructs. Moreover, the focus of this section also lies on constraints that frequently occur in benchmarks.

The easiest PB-constraints belonging to this category are those with two terms, since they correspond to quite easy Boolean constructs. However, here are only frequently occurring cases considered, so only the three PB-constraints are encoded as propositional formulas

$$-n \cdot x_1 + n \cdot x_2 \geq b \rightsquigarrow \begin{cases} \overline{x}_1 \wedge x_2, & b = n \\ true, & b = -n \\ x_1 \rightarrow x_2, & b = 0 \end{cases} \tag{3.17}$$

It is assumed that n is positive. In the first case b is equal to n. This constraint will only be satisfied, when x_1 is assigned to *0* and x_2 to *1*. Hence, it is encoded as $\overline{x}_1 \wedge x_2$. In the second case b is equal to *-n*. Since the smallest coefficient is equal to *-n* and the right hand side is negative, the constraint is always satisfied. Thus, it is encoded as *true*. In the last case b is equal to 0. The only case where the constraint is not satisfiable, is when only x_1 is assigned to *1*. Therefore, it must be guarantee that if x_1 is assigned to *1*, x_2 is also assigned to *1*. This corresponds to an implication.

Now, PB-constraints with three terms on the left hand side are considered. Basically, either one coefficient or two coefficients are negative. The other cases were already explained in section 3.2.2. First, constraints with one negative coefficient are regarded.

$$-n \cdot x_1 + n \cdot x_2 + n \cdot x_3 \geq b \rightsquigarrow \begin{cases} true, & b = -n \\ \overline{x}_1 \vee x_2 \vee x_3, & b = 0 \\ (x_1 \rightarrow (x_2 \wedge x_3)) \wedge (x_1 \vee x_2 \vee x_3), & b = n \end{cases} \tag{3.18}$$

In the first case b is equal to *-n* (n is positive). Since *-n* is the smallest and the only negative coefficient on the left hand side, there exists no assignment such that the sum would be less than *-n*. Hence, such constraints are encoded as *true*. In the next case, the right hand side is equal to 0. The assignment where only x_1 is assigned to *1* must be excluded. Therefore, as soon as x_1 is assigned to *1*, at least one of the other literals must also be set to *1*. Hence, x_1 implies that x_2 or x_3 are valid. The last case, is a little more complicated. The constraint requires that the sum on the left hand side is at least equal to n. Obviously, this constraint will not be satisfied when only x_1 is assigned to *1*. However, it can be satisfied when x_1 is set to *1* but only if x_2 and x_3 are also assigned to *1*. This is why the implication is needed. However, the implication itself is not enough because first, the cases where x_2 or x_3 are assigned to *1* are not satisfied by this formula. Second, the case when all literals are set to zero must also be excluded. This is why the disjunction over all coefficients is needed. It is unsatisfiable when all literals are assigned to *0*, but it is satisfied for cases when x_2 or x_3 are assigned to *1*. Hence, the conjunction over both sub-formulas gives a correct encoding for the origin PB-constraint.

Now, the cases where two coefficients are negative are discussed.

$$-n \cdot x_1 - n \cdot x_2 + n \cdot x_3 \geq b \rightsquigarrow \begin{cases} x_3, & b = n \\ \neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \rightarrow x3), & b = 0 \\ (x_1 \wedge x_2) \rightarrow x_3, & b = -n \end{cases} \tag{3.19}$$

The first case is quite clear since the sum on the left hand side must be at least n and on the left hand side two of the coefficients are negative. Clearly, the only possibility to satisfy the constraint is to assign x_3 to *1*. For all the other combinations, the constraint is unsatisfiable. The PB-constraint in the second case requires that the sum on the left hand side is equal to 0. The first sub-formula ensures, that x_1 and x_2 are never assigned to *1* at once. This case must be excluded, as the sum on the left hand side would already be equal to -2n and since -2n + n is less than zero, the constraint would not be satisfied. However, the negation of the sub-formula is for all the cases where x_1 and x_2 are not assigned to *1* satisfied. This means that this sub-formula by itself, is not satisfiability-equivalent to the PB-constraint. Therefore, the second sub-formula is needed. It guarantees that if x_1 or x_2 is assigned to *1*, x_3 is also assigned to *1*. Thus, the sum on the left hand side is always at least 0 and this is what the origin PB-constraint requires. In the last case b is equal to -n. Since there are two negative coefficients, one has to pay attention that not both literals belonging to the coefficients are set to *1*. Actually, this is what the implication guarantees. Since this is the only case which must be excluded, the formula is satisfiability-equivalent to the PB-constraint.

Obviously, there are much more cases which are not considered here. All of them are encoded as integer arithmetic formulas. How exactly the encoding works, is explained in the next section.

3.3 Encoding as Integer Arithmetic Formula

This section gives detailed information about the translation of PB-constraints into satisfiability-equivalent integer arithmetic formulas.

PB-constraints which must be encoded as integer arithmetic formulas will be solved by Simplex. This implies three things. First, in order to encode a PB-constraint as an integer arithmetic formula one has to change the type of the variables from Boolean to integer. Second, one has to guarantee that the new variables only takes the values *0* or *1*. Third, the new variables must be connected with the Boolean, as each Boolean variable and its corresponding integer variable must be assigned to the same value. The solution for this problems is to add new variables and to construct auxiliary constraints which guarantee that the mentioned problems does not occur. Using an example we explain how all three problems can be solved.

Consider the following two PB-constraints

$$2 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 + 1 \cdot x_4 + 1 \cdot x_5 \geq 3 \tag{3.20}$$

$$1 \cdot x_1 + 1 \cdot x_3 + 1 \cdot x_6 \geq 2 \tag{3.21}$$

The preprocessor first considers the first constraint. It belongs to the category of long formulas. Since the sum over coefficients on the left hand side is greater than the right hand side, the constraint should be encoded as an integer arithmetic formula

according to 3.2.2. As Simplex works on integer variables first, new integer variables i_1, i_2, i_3, i_4 and i_5 are created. Each Boolean variable is uniquely connected to one of the new integer variables because the Boolean variables might occur in other PB-constraints (here 3.21). For this purpose the preprocessor has a list which contains all Boolean variables and its corresponding integer variables. Here, the list looks like this:

$$\{(x_1, i_1), (x_2, i_2), (x_3, i_3), (x_4, i_4), (x_5, i_5)\} \tag{3.22}$$

Now the integer arithmetic formula can be created

$$2 \cdot i_1 + 2 \cdot i_2 + 3 \cdot i_3 + 1 \cdot i_4 + 1 \cdot i_5 \geq 3. \tag{3.23}$$

This formula could already be processed by Simplex. However, it must be satisfiability-equivalent to the formula 3.20. Until now, Simplex is allowed to assign the literals i_j to arbitrary integer values. Obviously, an assignment may be invalid for PB-constraints and so the formula 3.23 is not satisfiability-equivalent to the formula 3.20 yet. Moreover, it is possible that the SAT-solver already assigned some literals which also occur in this constraint. Therefore, auxiliary constraints are added. Those guarantee that if a Boolean variable is set to *true* (*false*) then the corresponding integer variable is set to *1* (*0*). Note that this constraint also ensures that the integer variables only take the values 0 or 1. The corresponding auxiliary constraint has the form

$$\bigwedge_{j=1}^{5} \Big((x_j \rightarrow (i_j = 1)) \wedge (\neg x_j \rightarrow (i_j = 0)) \Big) \tag{3.24}$$

The final translation of the formula 3.20 is

$$\psi := 2 \cdot i_1 + 2 \cdot i_2 + 3 \cdot i_3 + 1 \cdot i_4 + 1 \cdot i_5 \geq 3 \wedge \bigwedge_{j=1}^{5} \Big((x_j \rightarrow (i_j = 1)) \wedge (\neg x_j \rightarrow (i_j = 0)) \Big) \tag{3.25}$$

Now, the second constraint 3.21 is translated. Since 3.20 does not contain the Boolean variable x_6, a new integer variable i_6 is created and the variable list must be updated as follows

$$\{(x_1, i_1), (x_2, i_2), (x_3, i_3), (x_4, i_4), (x_5, i_5), (x_6, i_6)\}. \tag{3.26}$$

Since the auxiliary constraints for the variables i_1 to i_5 have already been added, only an auxiliary constraint for the new integer variable is added. This means that the formula 3.21 is encoded as

$$1 \cdot i_1 + 1 \cdot i_2 + 1 \cdot i_6 \geq 2 \wedge (x_6 \rightarrow (i_6 = 1)) \wedge (\neg x_6 \rightarrow (i_6 = 1)) \tag{3.27}$$

At this point the preprocessor forwards the conjunction over both translations 3.25 and 3.27 to the SMT-solver.

In general, a PB-constraint $\sum_j a_j \cdot x_j \ \# \ b$ is satisfiability-equivalent to the integer arithmetic formula

$$\sum_j a_j \cdot i_j \ \# \ b \ \wedge \ \bigwedge_j (x_j \rightarrow i_j = 1 \vee \overline{x}_j \rightarrow i_j \ = \ 0) \tag{3.28}$$

where x_j are Boolean variables and i_j are integer variables.

Chapter 4

Simplifying Pseudo-Boolean Constraints

Encoding large formulas can lead to exponential number of clauses in corresponding propositional formula or a large number of auxiliary formulas for integer arithmetic formulas. This chapter presents two approaches which can simplify PB-constraints and so accelerate solving of PB-constraints in some cases. In the first section an approach based on residual number systems is presented. In the second section it is shown how Gauss algorithm can be used to reduce the number of constraints and terms.

4.1 Simplifying Pseudo-Boolean Constraints Using Residual Number Systems

4.1.1 General Approach

This approach is based on *Chinese Remainder Theorem*. By means of this theorem, each integer can be uniquely decomposed into a sequence of smaller integers according to some base. Depending on the choice of the base, the sequence of integers can be longer or shorter. The mathematical operations addition, subtraction and multiplication can be applied component-by-component. The idea of this approach is to find a *residual number base* [FC14] which allows to represent each coefficient occurring in a PB-constraint by a minimal sequence of smaller integers. Once such a base is found, the PB-constraint can be translated to a conjunction of new PB-constraints which probably have smaller number of terms. The better the chosen base, the shorter the corresponding PB-constraints [FC14].

This approach is actually used by PB-solvers which encode all PB-constraints as Boolean formulas. However, for this PB-solver it should help to reduce the number of terms for some PB-constraints, such that they can be encoded as Boolean constraints [FC14].

4.1.2 Preliminaries

Mixed Radix Base. It is well-known that each natural number represented in decimal system can be represented in various numeral systems, e.g., binary system or hexadecimal system. Usually, each digit is represented using the same base. For example, in binary system the number 105 corresponds to 1101001 in binary system and the basis for each digit is $\langle 2,2,2,2,2,2,2 \rangle$. Basis where all digits are represented by the same integer are called *constant radix base*, where radices are the particular entries in the base. One can write this relationship as illustrated in table 4.1. However, it is possible to represent each digit using different radix. Such bases are called *mixed radix bases*. A finite mixed radix base is also a sequence of integers $\langle r_0, r_1, \ldots, r_k \rangle$, however the radices r_i does not have to be equal. Again, consider the number 105 and mixed radix base $\langle 2, 3, 4, 5 \rangle$. The representation of this number is illustrated in 4.2. The digits can be calculated as follows

$$\begin{aligned} d_0 &: \ 105 \ mod \ 2 = 1 \\ d_1 &: \ 52 \ mod \ 3 = 1 \\ d_2 &: \ 17 \ mod \ 4 = 1 \\ d_3 &: \ 4 \ mod \ 5 = 4 \end{aligned}$$

Definition 4.1.1 (Chinese Remainder Theorem)**.** *Let $n_1, n_2, \ldots n_k$ be a sequence of pairwise coprime integers, then there exists exactly one integer n ($0 \leq n < \prod_i n_i$) which can be represented by a sequence of integers $a_1, a_2, \ldots, a_k$ ($0 \leq a_i \leq n_i$ for each i) such that*

$$n \equiv a_i \ (mod \ n_i) \tag{4.1}$$

holds for all $i \in \{1, \ldots, k\}$ [CLRS09].

An important conclusion of this theorem is that the mathematical operations addition, subtraction and multiplication which are applied on two integers x and y can also be applied on their decomposition. This means if x can be decomposed into the sequence of integers $(x_1, \ldots, x_k)$ and y can be decomposed into the sequence $(y_1, \ldots, y_k)$, then the following holds ($z \in \mathbb{N}$)

$$\begin{aligned} (x+y) \ mod \ z &\iff ((x_1+y_1) \ mod \ z, (x_2+y_2) \ mod \ z, \ldots, (x_k+y_k) \ mod \ z), \\ (x-y) \ mod \ z &\iff ((x_1-y_1) \ mod \ z, (x_2-y_2) \ mod \ z, \ldots, (x_k-y_k) \ mod \ z), \\ (x \cdot y) \ mod \ z &\iff ((x_1 \cdot y_1) \ mod \ z, (x_2 \cdot y_2) \ mod \ z, \ldots, (x_k \cdot y_k) \ mod \ z). \end{aligned} \tag{4.2}$$

Radix	**2**	**2**	**2**	**2**	**2**	**2**	**2**
Digits	d_0	d_1	d_2	d_3	d_4	d_5	d_6
105	1	1	0	1	0	0	1

Table 4.1: Representation of 105 in constant radix base.

Radix	**2**	**3**	**4**	**5**
Digit	d_0	d_1	d_2	d_3
105	1	1	1	4

Table 4.2: Representation of 105 using mixed radix base $\mu = \langle 2, 3, 4, 5 \rangle$.

Residual Number Base. A *residual number (RNS) base* is a sequence of pairwise coprime integers greater one, called *moduli*. An RNS base is a base for a PB-constraint if additionally, the product over the moduli is greater than the maximum coefficient

occurring in the PB-constraint and the number of moduli is minimal. However, not all PB-constraints have an RNS base which is non-redundant. For example cardinality constraints have no non-redundant RNS bases, as 1 modulo any number again results in 1. This means that the calculation of bases for such constraints is redundant.

Due to the Chinese remainder theorem, every integer x can uniquely be represented in an RNS base $\mu = \langle m_0, m_1, \ldots, m_n \rangle$ as follows

$$x_\mu = \langle (x \ mod \ m_0), (x \ mod \ m_1), \ldots, (x \ mod \ m_n) \rangle$$

Thus, given the RNS base $\mu = \langle 2,3,5,7,11 \rangle$, the number 748 represented in μ has the form $(748)_\mu = (0, 1, 3, 6, 0)$, since

$$\begin{aligned} 748 \ mod \ 2 &= 0 \\ 748 \ mod \ 3 &= 1 \\ 748 \ mod \ 5 &= 3 \\ 748 \ mod \ 7 &= 6 \\ 748 \ mod \ 11 &= 0 \end{aligned}$$

Since addition, subtraction and multiplication can be performed on the numbers represented in an RNS base without any restriction, this representation can be used for simplification of PB-constraints. In doing so, each coefficient of a PB-constraint must first be represented in the base. Out of this representation an equations system of *pseudo-Boolean modulo (PB-Mod)* constraints can be derived. The following example illustrates how the simplification works. The following constraint is considered

$$748 \cdot x_1 + 936 \cdot x_2 + 58 \cdot x_3 + 493 \cdot x_4 + 145 \cdot x_5 + 85 \cdot x_6 = 105 \tag{4.3}$$

Digit	d_0	d_1	d_2	d_3	d_4
Radix	**2**	**3**	**5**	**7**	**11**
748	0	1	3	6	0
935	1	2	0	4	0
58	0	1	3	2	3
493	1	1	3	3	9
145	1	1	0	5	2
85	1	1	0	1	8
105	1	0	0	0	6

Table 4.3: Representation of the coefficients from equation 4.3 in the RNS base $\mu_1 = \langle 2, 3, 5, 7, 11 \rangle$

Digit	d_0	d_1	d_2
Radix	**5**	**17**	**29**
748	3	0	23
935	0	0	7
58	3	7	0
493	3	0	0
145	0	9	0
85	0	0	27
105	0	3	18

Table 4.4: Representation of the coefficients from equation 4.3 in the RNS base $\mu_2 = \langle 5, 17, 29 \rangle$

The tables 4.3 and 4.4 show the coefficients occurring in the equation 4.3 represented in the two different RNS bases $\mu_1 = \langle 2, 3, 5, 7, 11 \rangle$ and $\mu_2 = \langle 5, 17, 29 \rangle$. The first base ($\mu_1$) yields the following equations system

$$\begin{aligned} 1 \cdot x_2 + 1 \cdot x_4 + 1 \cdot x_5 + 1 \cdot x_6 &= 1 \ mod \ 2 \\ 1 \cdot x_1 + 2 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_4 + 1 \cdot x_5 + 1 \cdot x_6 &= 0 \ mod \ 3 \\ 3 \cdot x_1 + 3 \cdot x_3 + 3 \cdot x_4 &= 0 \ mod \ 5 \\ 6 \cdot x_1 + 4 \cdot x_2 + 2 \cdot x_3 + 3 \cdot x_4 + 5 \cdot x_5 + 1 \cdot x_6 &= 0 \ mod \ 7 \\ 3 \cdot x_3 + 9 \cdot x_4 + 2 \cdot x_5 + 8 \cdot x_6 &= 6 \ mod \ 11 \end{aligned} \tag{4.4}$$

The second base (μ_2) yields this equations system

$$\begin{aligned} 3 \cdot x_1 + 3 \cdot x_3 + 3 \cdot x_4 &= 0 \ mod\ 5 \\ 7 \cdot x_3 + 9 \cdot x_5 &= 3 \ mod\ 17 \\ 23 \cdot x_1 + 7 \cdot x_2 + 27 \cdot x_6 &= 18 \ mod\ 29 \end{aligned} \tag{4.5}$$

Obviously, the second equations system is shorter than the first one. This may mean, that the second equations system can be solved faster than the first one. This implies that the choice of base is important. However, the resulting constraints are so called *pseudo-Boolean modulo (PB-Mod) constraints* which cannot directly be processed by the preprocessor. Hence, one needs an encoding for them [FC14].

4.1.3 Encoding Pseudo-Boolean Modulo Constraints

In order to simplify a PB-constraint, a non-redundant RNS base $\mu = \langle p_1, \ldots, p_m \rangle$ was used. The result was a conjunction of PB-Mod-constraints. Now, since the preprocessor should be used to solve the conjunction, the PB-Mod-constraints must be transformed into regular PB-constraints. For the modulo operation it holds that if $a \ mod\ b = c$ then there exists an integer t such that it holds $t \cdot b + c = a$. This can be applied on the PB-Mod-constraints

$$\bigwedge_{j=1}^{m} \left(\sum_{i=1}^{n} (a_i \ mod\ p_j) \cdot x_i \equiv c_j \ mod\ p_j \right) \Leftrightarrow \bigwedge_{j=1}^{m} \left(\sum_{i=1}^{n} (a_i \ mod\ p_j) \cdot x_i = t_j \cdot p_j + c_j \right) \tag{4.6}$$

However, t_j is an arbitrary integer. According to the definition of the modulo operation for every t_j holds

$$t_j = \left\lfloor \frac{(\sum_{i=1}^{n} (a_i \ mod\ p_j) \cdot x_i) - c_j}{p_j} \right\rfloor \leq \left\lfloor \frac{(\sum_{i=1}^{n} (a_i \ mod\ p_j)) - c_j}{p_j} \right\rfloor =: k \tag{4.7}$$

Hence, the value of t_j can be bounded by k. Thus, each variable t_j can be encoded using unary bit-blasting, that means

$$t_j = \sum_{i=0}^{k} y_{ij} \tag{4.8}$$

where y_{ij} is a variable that can either take the value *0* or *1* [FC14].

All in all, a PB-Mod-constraint can be transformed into a regular PB-constraint by

$$\sum_{i=1}^{n} (a_i \ mod\ p_j) \cdot x_i \equiv c_j \ mod\ p_j \Leftrightarrow \sum_{i=1}^{n} (a_i \ mod\ p_j) \cdot x_i - p_j \cdot (\sum_{i=0}^{k} y_{ji}) = c_j \tag{4.9}$$

For example, the PB-Mod-constraint $23 \cdot x_1 + 7 \cdot x_2 + 27 \cdot x_6 = 18 \ mod\ 29$ is equivalent to the PB-constraint $23 \cdot x_1 + 7 \cdot x_2 + 27 \cdot x_6 - 29 \cdot y_{11} = 18$, as $\lfloor \frac{23+7+27-18}{29} \rfloor = 1$ and so only one additional variable is required [FC14].

4.1.4 Finding Optimal Residual Number Base

As seen in the previous subsection, the choice of base influences the number of variables occurring in the equation system after RNS-transformation. In order to minimize the number of occurring variables it is important to choose a base that contains primes which divide the maximum number of coefficients [FC14].

Coefficient	**Prime Factors**
748	2, 11, 17
935	5, 11, 17
58	2, 29
493	17, 29
145	5, 29
85	5, 17

Table 4.5: Prime factors of the coefficients of equation 4.3.

Definition 4.1.2. *A non-redundant RNS base μ is called optimal if and only if there exists no other RNS base whose moduli divides more coefficients than μ.*

In order to find such a base first, one has to perform prime factorization. Once, the prime factorization is performed for all coefficients, the primes are sorted first lexicographic, and then according to the number of divided coefficients. The shortest sequence of the sorted primes whose product is greater than the maximal coefficient is the optimal base. An example should illustrate the approach [FC14].

Again the equation 4.3 is considered. First the prime factorization is performed. Table 4.5 shows the resulting prime factors. Now for each prime factor, the number of divided coefficients can be count. The next step is to sort the primes according to

Prime Factor	**2**	**5**	**11**	**17**	**29**
Number of divided coefficients	2	3	2	4	3

Table 4.6: Number of coefficients divided by the corresponding prime factor.

the number of divided coefficients. The order of the primes is $(17, 5, 29, 2, 11)$. The optimal RNS base is the shortest sequence of integers whose product is greater than 935. As $17 \cdot 5$ is smaller than 935 and $17 \cdot 5 \cdot 29$ is greater than 935, the sequence $\langle 17, 5, 29 \rangle$ is the optimal RNS base [FC14].

4.2 Gauss Algorithm for Simplifying Pseudo-Boolean Constraints

4.2.1 General Approach

A conjunction of PB-constraints with the relation $=$ can be interpreted as a system of regular equations on which the well-known *Gauss algorithm* can be performed. We construct a matrix M that contains all the left hand sides of the equations. Afterwards, Gauss algorithm is performed on it. The resulting matrix M' can be used to simplify the inequalities. In best case, this procedure can reduce the number of constraints which have to be processed by the SMT-solver.

Preparing Equations. First, a matrix M that contains all the left hand sides of the equations is constructed. Afterwards, the *lower upper (LU) decomposition* can be performed on this matrix [Eig17]. The decomposition decomposes the matrix into a

unit-lower-triangular matrix L and an *upper-triangular* matrix U. The upper matrix U will later on be used to simplify the remaining constraints. Note that the actual implementation of the LU decomposition also performs permutations on the matrix. The right hand side must therefore be adapted according to these permutations [Eig17].

Simplifying Inequalities. An upper matrix can be used to simplify inequalities. In doing so, a matrix I is constructed which contains all the left hand sides of the inequalities. Now, for each row i in the matrix I, one look for a row in matrix U which can make an entry in row i equal to 0. This is done as long as there exists no more row in the upper matrix which could make any entry in row i equal to 0. Once all rows in matrix I are simplified, one can transform both matrices (I and U) to PB-constraints. Finally, the resulting constraints are forwarded to the preprocessor which can encode them.

The following example illustrates how the approach works. The following PB-constraints are considered

$$\begin{aligned} 1 \cdot x_1 - 2 \cdot x_2 + 1 \cdot x_3 &= 4 \\ 2 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 &= 5 \\ 1 \cdot x_1 + 1 \cdot x_2 &= 2 \\ -5 \cdot x_2 + 1 \cdot x_3 &\geq 2 \\ 4 \cdot x_1 + 1 \cdot x_2 + 4 \cdot x_4 &\geq 1 \end{aligned} \tag{4.10}$$

The matrix M representing the equations has the form

$$M = \begin{pmatrix} 1 & -2 & 1 \\ 2 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \tag{4.11}$$

Now, the upper matrix U can be calculated. However, our approach works on integers so the upper matrix must be transformed in such a way that all elements are integers. Thus, the upper matrix has the form

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -5 & 1 \\ 0 & 0 & -2 \end{pmatrix} \tag{4.12}$$

For the right hand side holds b = (5, 2, -1). Matrix I representing the inequalities has the form

$$I = \begin{pmatrix} 0 & -5 & 1 \\ 4 & 1 & 4 \end{pmatrix} \tag{4.13}$$

Using matrix U one can simplify matrix I such that the following matrix results

$$I' = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix} \tag{4.14}$$

Hence, the following PB-constraints can be produced

$$\begin{aligned} 2 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 &= 5 \\ -5 \cdot x_2 + 1 \cdot x_3 &= 2 \\ -2 \cdot x_3 &= -1 \\ -1 \cdot x_2 &\geq -10 \end{aligned} \tag{4.15}$$

In this case the Gauss algorithm could reduce the number of constraints as well as the number of terms. Usually, this should accelerate the solving of PB-constraints.

Chapter 5

Experimental Results

The approaches presented in this thesis were implemented using the modular C++ library *SMT-RAT* [CKJ+15]. SMT-RAT offers various *modules* which implement different SMT solving procedures, such as SAT-solver or Simplex-solver. The different modules can be used to compose a user-defined SMT-solver. For this purpose, one has to define a *strategy*. In a strategy a user can define of which modules a SMT-solver is composed and in which order the modules should be executed. Moreover, one can define which parts of code are executed or not.

The SMT-solver used in this thesis combines the SAT-solver and the Simplex-solver with a newly implemented preprocessor module for PB-constraints. In order to test the various encoding and simplification approaches, different strategies were created. All strategies implement the encoding of arithmetic formulas, short and long formulas with consistent signs. Overall six strategies were established. Table 5.1 gives an overview over all strategies. The first column contains the names of the strategies. The next two columns show if a strategy encodes cardinality constraints or long formulas with non-consistent sings as Boolean formulas. The last two columns indicate if Gauss or RNS procedure are used by a strategy. At this point it is important to underline that all optimization possibilities were ignored.

Strategy	Cardinality Constraints	Long Formulas with Non-Consistent Signs	Gauss Algorithm	RNS
Basic	✗	✗	✗	✗
Cardinality Constraints	✓	✗	✗	✗
Non-Consistent Signs	✗	✓	✗	✗
Gauss	✓	✓	✓	✗
RNS	✓	✓	✗	✓
Complete	✓	✓	✗	✗

Table 5.1: Overview over strategies for SMT-RAT.

The benchmarks are taken form the pseudo-Boolean Competition 2015 and 2016. Each test was run with a timeout of 30 seconds. As a reference the PB-solver *MiniSat+*

[ES06] was taken which encodes all PB-constraints as Boolean formulas. Overall 4597 benchmarks were run. SMT-RAT was able to solve about 20% of the overall number of examples. MiniSat+ was able to solve about 60% of the examples. Table 5.2 shows how many examples a certain strategy was able to solve. Thus, SMT-RAT was overall

Strategy	Number of solved examples
Basic	913
Cardinality Constraints	677
Non-Consistent Signs	1023
Gauss	766
RNS	829
Complete	829

Table 5.2: Number examples solved by the different strategies.

worse than MiniSat+. However, there are some benchmark classes where SMT-RAT is much faster than MiniSat+. Those are presented in this chapter.

In order to show how fast SMT-RAT was able to solve different problems, various diagrams have been made. The x-axis of each diagram shows the overall number of variables occurring in an example. The y-axis shows the time needed for solving the problem in milliseconds.

The first section deals with the comparison of MiniSat+ and SMT-RAT. The second section deals with the comparison of the different strategies for SMT-RAT.

5.1 Comparison of MiniSat+ and SMT-RAT

The diagrams presented in this section show the results for two different benchmark classes. The first class contains problems which are similar to the already mentioned subset sum problem. Those are PB-constraints whose sum on the left hand side is greater than the right hand side. Thus, according to chapter 3 all of them are encoded as arithmetic formulas. The second class contains problems which are represented by a sequence of long PB-constraints with consistent signs, which SMT-RAT encodes as Boolean constraints. However, there is always one formula containing all variables which is encoded as arithmetic formula.

Here, SMT-RAT always used the basic strategy. It turned out that SMT-RAT usually becomes better with increasing number of variables for the two benchmark classes. This behaviour is presented by the diagrams below.

Figure 5.1 illustrates a diagram presenting the results for benchmarks belonging to the first class. The benchmarks were taken from the pseudo-Boolean Competition 2015 and belong to the group "Proof Complexity". The problems solved here were called "Fixed Bandwidth". One can see that only at the beginning MiniSat+ is faster than SMT-RAT. When the number of variables is greater than about 45, SMT-RAT can solve the problems faster than MiniSat+.

A similar behaviour can be observed in figures 5.2 and 5.3. Both figures present benchmarks belonging to the same class and group. The benchmarks presented in figure 5.2 show problems called "Regular Extracted". Benchmarks presented in figure

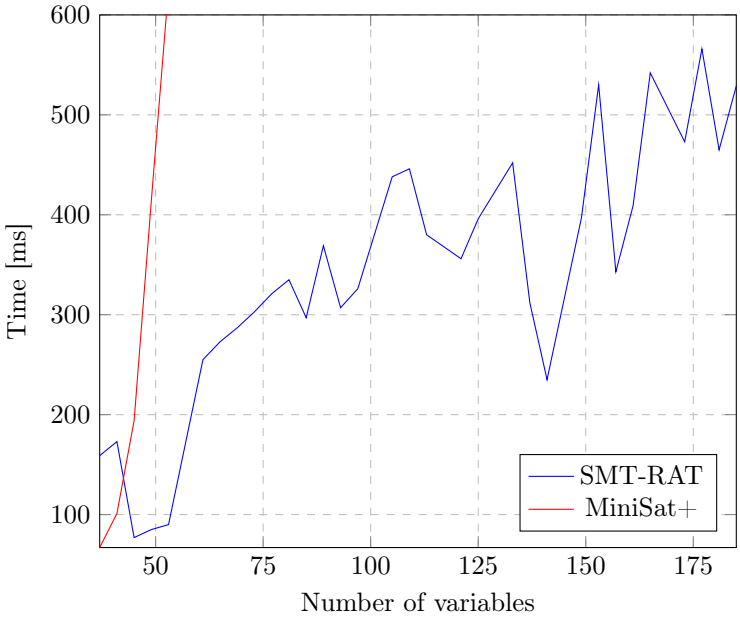

Figure 5.1: Results for the benchmarks belonging to the first class. The problems solved here are called "Fixed Bandwidth".

5.3 were called "Subset Cardinality Sigma". At the beginning in both cases SMT-RAT is worse than MiniSat+ however, when the number of variables reaches a certain mark, SMT-RAT can solve the problems faster. This show that when problems belonging to this class reach a certain complexity, arithmetic encoding of the constraints is better than Boolean encoding.

Figure 5.4 shows results for benchmarks belonging to the second benchmark class. The benchmarks used here are from year 2016 and belong to the group "Vertexcover Plain". Here, one can clearly see that for this benchmark group SMT-RAT is much better than MiniSat+. Similar behaviour illustrates figure 5.5. The benchmars showed here also belong to the same class. However, they belong to the group "Vertexcover Hard". One can observe that again at the beginning MiniSat+ is slightly better than SMT-RAT. This holds for examples consisting of at most 110 variables. Afterwards, the performance of MiniSat+ vary strongly. However, it is unable to solve problems consisting of about 170 variables within 30 seconds.

All in all, one can see that for both classes arithmetic encoding of PB-constraints can pay off. Especially when the number of variables in such PB-constraints is very high. The reason for this behaviour is probably the fact, that Simplex is usually faster than a bit vector procedure which is used by MiniSat+.

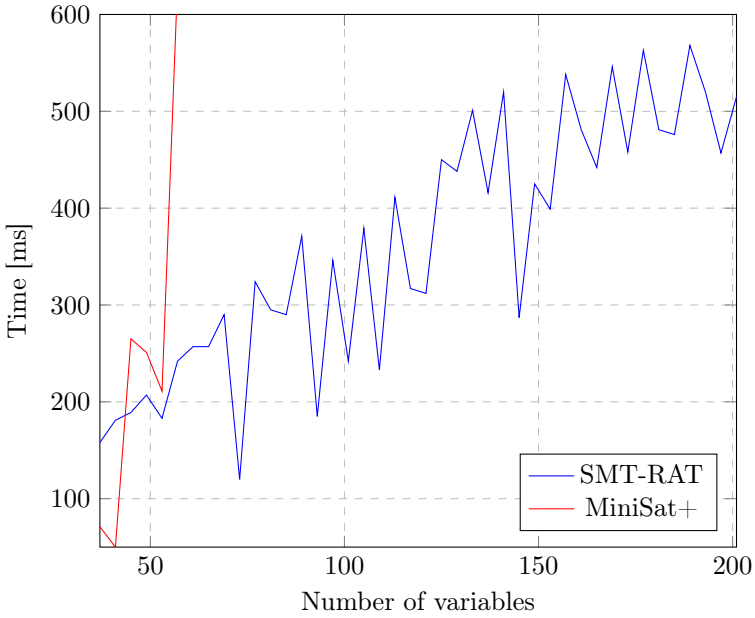

Figure 5.2: Results for the benchmarks belonging to the first class. The problems solved here are called "Regular Extracted".

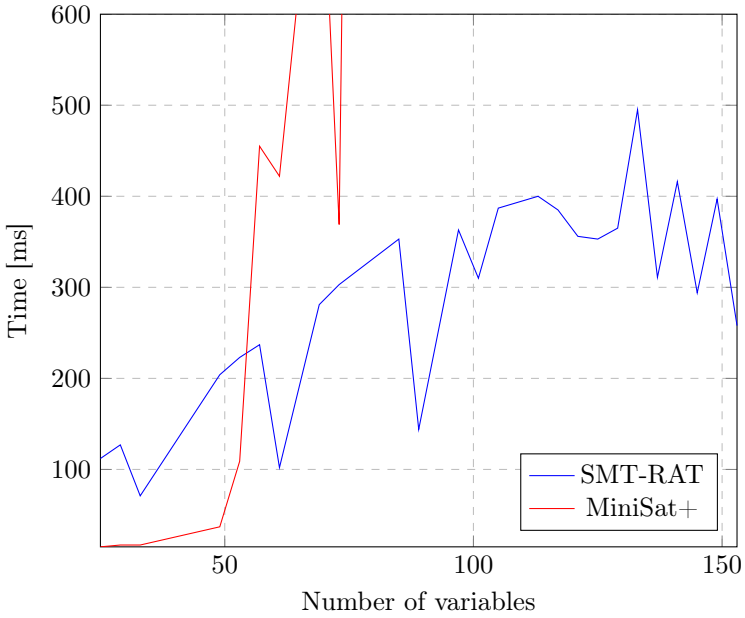

Figure 5.3: Results for the benchmarks belonging to the first class. The problems solved here are called "Subset Cardinality Sigma".

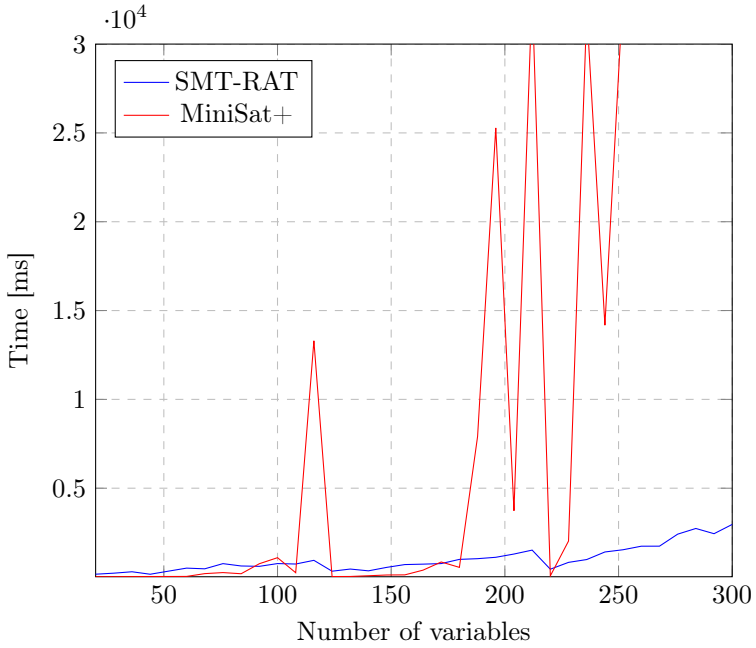

Figure 5.4: Results for the benchmarks belonging to the second class. The problems solved here are called "Vertexcover Plain".

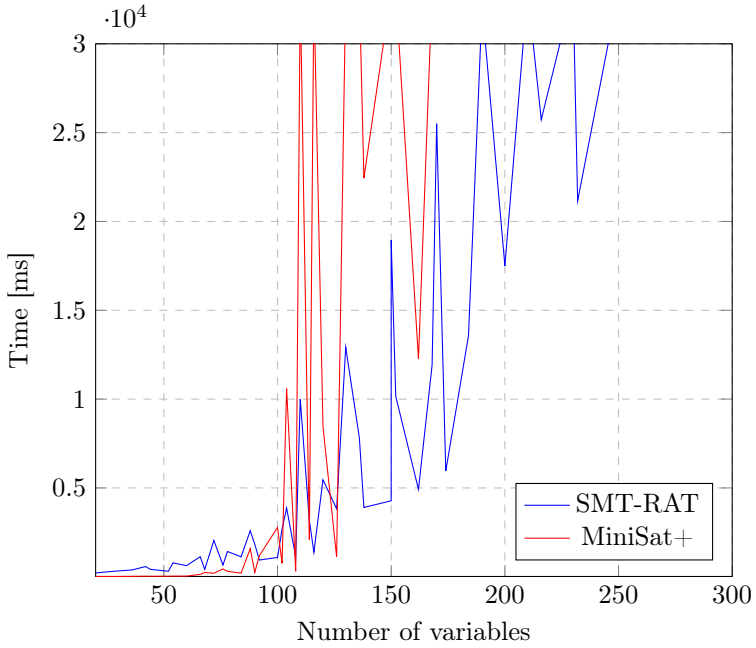

Figure 5.5: Results for the benchmarks belonging to the second class. The problems solved here are called "Vertexcover Hard".

5.2 Comparison of Strategeis for SMT-RAT

This section compares the different strategies for SMT-RAT. Each diagram compares the `basic` strategy with strategy which was the best.

Figure 5.6 shows the results for benchmarks from year 2015 belonging to the grout "*EC_ODD_GRIDS*". All constraints were equations with positive left and right hand side. However, the right hand side is always smaller than the sum over coefficients on the left hand side. Thus, according to chapter 3, all constraints were encoded as Boolean constraints by the strategies `cardinality constraints` and `complete`. However, `RNS` could not accelerate the solving, since all coefficients in this class are either equal to 1 or to -1. Actually, it turned out that the strategies `cardinality constraints` and `complete` are much better than the `basic` strategy. The diagram shows only the `cardinality constraints` strategy, since the results for both strategies were very similar. One can clearly see, that the `cardinality constraint` strategy is much better even when the number of variables reaches the 600 mark.

Another interesting benchmarks are those from year 2015 belonging to the group "Proof Complexity". The problems are called "Regular Plain" (look Fig. 5.7). This problems are similar to those mentioned before ("Regular Extracted"). The best strategies were `cardinality constraints`, `Gauss`, `complete` and `RNS`. Since the constraints occurring in this examples are only inequalities, RNS was not applicable. However, the `RNS` strategy encodes cardinality and long formulas with non-consistent signs. Thus, RNS did not accelerate anything. The same holds for `Gauss` strategy. The diagram shows the results of `complete` strategy, as all the mentioned strategies had similar results. The `basic` strategy was not able to solve even one example while the `complete` strategy became worse when the number of variables became about 85.

All in all there are several classes of problems which SMT-RAT can solve much faster than MiniSat+. The arithmetic encoding especially pays off for PB-constraints with large number of terms. However, for small PB-constraints Boolean encoding seems to be better. Unfortunately, both simplifying approaches seem not to work well. According to [FC14], RNS is suitable for long constraints and works well when they are encoded as Boolean constraints. In our approach, in most cases long formulas are solved by Simplex that works well with big integers. Gauss algorithm does not help probably due to similar reasons. However, overall one can say that arithmetic encoding of PB-constraints is sensible and can accelerate the solving process.

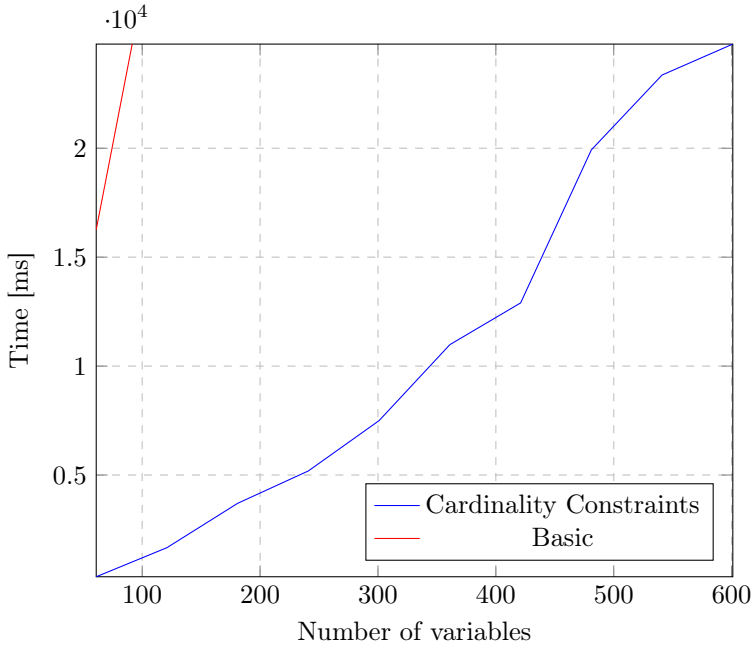

Figure 5.6: Results for the benchmarks from year 2015 belonging to the group "*EC_ODD_GRIDS*".

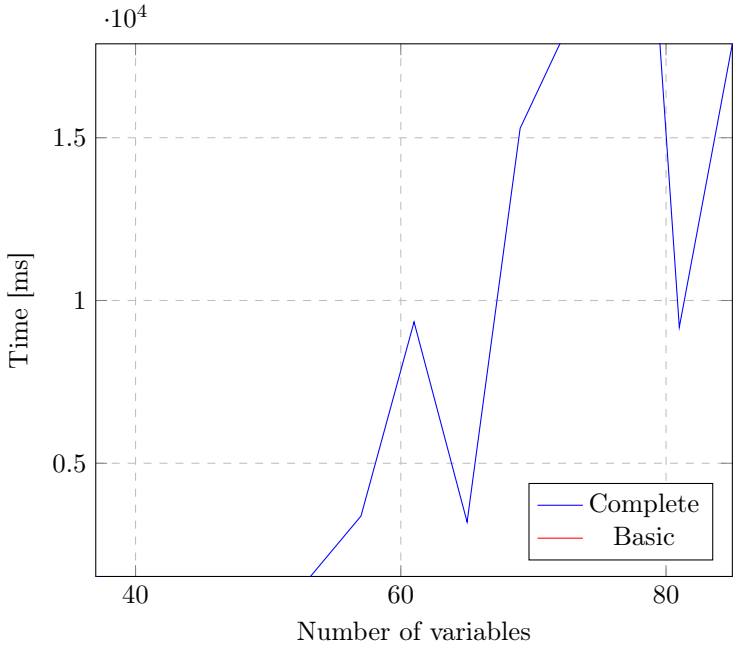

Figure 5.7: Results for the benchmarks from year 2015 belonging to the group "Proof Complexity". The problems are called "Regular Plain".

Chapter 6

Conclusion

This thesis proposed an SMT-solver for solving PB-constraints. The basic idea was to transform every PB-constraint into either a Boolean formula and an arithmetic constraint and combine these into a linear integer arithmetic formula. In order to solve this formula a SMT-solver was used. In order to accelerate the encoding two approaches for simplifying PB-constraints were proposed. One approach used residual number bases in order to reduce the number of terms occurring in a PB-constraint. The other approach implemented Gauss algorithm which reduced the overall number of PB-constraints. Using SMT-RAT the approach proposed here could be compared to one of the most popular PB-solvers MiniSat+. The experimental results showed that arithmetic encoding of PB-constraints can pay off when the number of variables grows. However, both simplification approaches seemed not to help.

Future work might consider non-linear PB-constraints and include the possibility to add objective functions. Moreover, more detail analysis of the PB-constraints could probably lead to more efficient Boolean encoding.

Bibliography

[BA12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer Publishing Company, Incorporated, 3rd edition, 2012.

[Bar96] Peter Barth. *Logic-Based 0-1 Constraint Programming*. Springer US, 1996.

[BBH+09] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[BH02] Endre Boros and Peter L. Hammer. Pseudo-boolean optimization. *Discrete Appl. Math.*, 123(1-3):155–225, November 2002.

[CESS08] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. Sat-solving in practice. In *2008 9th International Workshop on Discrete Event Systems*, pages 61–67, May 2008.

[CK03] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 830–835, June 2003.

[CK05] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, March 2005.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. *SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving*, pages 360–368. Springer International Publishing, Cham, 2015.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Eig17] Eigen. Eigen::FullPivLU< MatrixType > Class Template Reference. `https://eigen.tuxfamily.org/dox/classEigen_1_1FullPivLU.html`, 2017. [Online; accessed 15-June-2017].

[ES06] Niklas EÃľn and Niklas SÃűrensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[FC14] Yoav Fekete and Michael Codish. *Simplifying Pseudo-Boolean Constraints in Residual Number Systems*, pages 351–366. Springer International Publishing, Cham, 2014.

[Kra95] Jan Krajíček. *Bounded Arithmetic, Propositional Logic, and Complexity Theory*. Cambridge University Press, New York, NY, USA, 1995.

[Seb07] Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying uml/ocl models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1341–1344, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[Woe03] Gerhard J. Woeginger. *Combinatorial Optimization - Eureka, You Shring!* Springer Berlin Heidelberg, 2003.