SATISFIABILITY OF QUANTIFIED BIT-VECTOR FORMULAS THEORY & PRACTICE

MARTIN JONÁŠ

PHD THESIS
May 2019

Faculty of Informatics
Masaryk University

SUPERVISOR
doc. RNDr. Jan Strejček, Ph.D.

Martin Jonáš: *Satisfiability of Quantified Bit-Vector Formulas Theory and Practice,* PhD thesis, May 2019

He too concludes that all is well. This universe henceforth without a master seems to him neither sterile nor futile. Each atom of that stone, each mineral flake of that night filled mountain, in itself forms a world. The struggle itself toward the heights is enough to fill a man's heart. One must imagine Sisyphus happy.

— Albert Camus, *The Myth of Sisyphus*

ABSTRACT

Multiple modern applications, ranging from bioinformatics, through job scheduling, to software and hardware analysis rely on ability to decide satisfiability of a given formula in a given logical theory. This decision problem is called satisfiability modulo theories (SMT). This thesis focuses on one particular theory, which is of paramount importance in analysis of software and hardware: the theory of fixed-size bit-vectors. In particular, this thesis focuses on both theoretical and practical aspects of deciding satisfiability of bit-vector formulas that contain quantifiers.

In the first part of the thesis, we investigate the precise computational complexity of satisfiability of quantified bit-vector formulas in which the values of bit-widths are encoded in binary or decimal notation, which is often the case in practice. Afterwards, we introduce an approach to deciding satisfiability of quantified bit-vector formulas by using binary decision diagrams and abstractions of bit-vector operations. We also the extend known simplifications of formulas that contain unconstrained variables, i.e., variables that occur only once in the formula. The thesis further describes our implementation of state-of-the-art SMT solver called Q3B, which uses the approach of solving satisfiability of quantified bit-vector formulas by binary decision diagrams and abstractions and also implements the simplifications using unconstrained variables. Experimental evaluation shows that the implemented solver Q3B outperforms other state-of-the-art solvers for quantified bit-vector formulas. Furthermore, we also investigate the dependence of satisfiability of quantified bit-vector formulas on the bit-widths of the used variables. We show that in most formulas coming from practical applications, the satisfiability of the formula remains the same even after reducing the bit-widths in the formula to very small values. Finally, we show how to use this observation to improve the performance of quantified bit-vector SMT solvers by using reductions of bit-widths in the input formula.

ACKNOWLEDGEMENTS

First things first. I am most grateful to my father, Vlastimil, who had always inspired my interest in mathematics, science, and the world around me. Without you, this thesis would not exist. I wish you could see it.

The thesis also would not exist without other members of my family; in particular my mother, Lenka, and my sister, Marika. Thank you for always supporting me and teaching me to appreciate nature, not to take life too seriously, and to laugh at weird jokes that no-one else would laugh at. I would also like to thank the rest of my family for supporting me, namely to Pavel, Martina, František, Máňa, Milada, Oldřich, Lucie, Adéla, Natálie, Nikola, Roman, Miluška, Fanda, Lenka, and Lucie.

I am equally thankful to Viki for all the places we have seen, things we have done, and interesting discussions we have had. These past few years would not be nearly as pleasant without you.

The thesis also would not exist without my supervisor, Jan Strejček, to whom I am deeply indebted. He came up with the interesting subject of study and has taught me to conduct research, write clearly, prepare interesting presentations, and think both about the big picture and minute details. He also never hesitated to share his knowledge regarding both scientific topics and a multitude of unrelated topics such as chocolate, aviation, tennis, or architecture.

During my studies, I have been lucky to share an office with four extraordinary colleagues. I am thankful to Fanda for showing me how to care about teaching and explaining things to people. To Ľuboš for all the bizarre experiences during travels. To Dominik for teaching me to be humble by showing me how bad a mathematician I am. And to Kuba for teaching me the same thing by showing me how bad a logician and theoretical computer scientist I am.

I also had the opportunity to meet amazing scholars, researchers, and fellow students. I am grateful to Mojmír Křetínský for always being supportive and kind not only towards me, but towards all other people. To Vláďa for showing me that a theoretical computer scientist can also be a good teacher and an excellent programmer. To Heňo for all the discussions about typography, photography, art, and for all the experiences from our travels. To Honza for all the discussions about technology, movies, and pop-culture. To Adam for all the discussions about human languages. To Marťa and Marek for always being high-spirited. To Martin for trying to improve teaching skills at our faculty and for his organizing skills, which I am still striving to achieve. To Jiřík and Ivana for making my complaints on having too much work to do seem silly in comparison. To Nikola and Mornfall for always having been willing to share pieces of wisdom from incredibly diverse topics. To Vojta and Obďa for showing inspiring passion for teaching. Generally, I am grateful to all the members of the laboratories Formela and ParaDiSe for creating such a pleasant atmosphere to work in.

I am also grateful to all my friends, who have been an important part of my life. In particular, to Milan for all the discussions about technology, music, good movies, and especially about bad movies. To Ondra for all the discussions about politics, economy, and psychology. To both of you, together with Terka and Síma, for all the wonderful memories from our bowling years. To Andrejka for showing me that there are important things in life that I had been missing and for always being eager to discuss art, philosophy, literature, and life in general. To Pája and Michal for never hesitating to share interesting, weird, or slightly disgusting facts about the human body. To Michal for all the episodes of Yes Minister I had heard while we were roommates. To Anet for her inspiring passion for cooking and for often being the only real adult in the room. To Petja for always laughing at my stupid jokes. To Síma and David for showing me their enthusiasm for traveling and exploring the world. To my former bandmates Štepán, Šoufek, Miloš, and Tučkin for the incredible memories from rehearsals. To Tom, Katka, and their Buchta Coffee for supplying me with plenty of excellent coffee, without which this thesis would definitely not exist.

Finally, if we had a conversation together that was at least slightly beyond small talk, I am grateful to you as well. If it were not for this conversation, my life might be different and possibly not that good.

Martin Jonáš
Brno
May 2019

CONTENTS

1 INTRODUCTION

In the modern world, as computer software becomes still more ubiquitous and complex, there is an increasing need to test it and formally verify its correctness. Several approaches to software verification, such as symbolic execution or bounded model checking, rely on the ability to decide whether a given first-order formula in a suitable logical theory is satisfiable. To this end, many of the tools for software verification use Satisfiability Modulo Theories (SMT) solvers, which can solve precisely the task of checking satisfiability of a given first-order formula in a given logical theory. The logical theory describes the set of objects that can be assigned to variables and also describes the behaviour of operations and relations. However, classical theories such as the theory of integers or real numbers are not well suited for software verification. For example, consider the following program.

```
int x = read();
int y = read();
int z = read();
if (x != 0 && y != 0)
{
  print(z / (x * y));
}
```

The program contains division by zero precisely if the formula

$$x \neq 0 \quad \wedge \quad y \neq 0 \quad \wedge \quad x \times y = 0$$

is satisfiable. Although this formula is satisfiable neither for integers nor for real numbers, the division by zero can indeed happen in the program. This happens due to overflows, since the operations in the program are computed modulo 2^{32}.

Therefore, for describing software, the natural choice of a logical theory is the theory of *fixed-size bit-vectors*, in which the objects are vectors of bits and its operations and relations precisely reflect computations performed by computers. It is thus not surprising that quantifier-free bit-vector formulas are used in tools for symbolic execution, bounded model checking, analysis of hardware circuits, static analysis, or test generation [Cad+08; CDE08; Lei10; CFM12; Gad+18]. There exist multiple SMT solvers that can decide satisfiability of *quantifier-free* bit-vector formulas: for example Beaver [JLS09], Boolector [Nie+18a], CVC4 [Bar+11], MathSAT5 [Cim+13], mcBV [ZWR16], OpenSMT [Hyv+16], Sonolar [PVL11], Spear [Hut+07], STP [GD07], UCLID [LS04], Yices [Dut14], or Z3 [MB08].

However, for some applications, quantifier-free formulas are too restrictive or not expressive enough. For example, the formulas with quantifiers naturally arise in applications such as synthesis of invariants, ranking functions, or loop summaries, or in testing equality of two symbolically represented sets of states of a program [GSV09; SGF10; Coo+13; KLW15; Mrá+16]. It is therefore not surprising that recent years have seen several research advances in the field of solving satisfiability of *quantified* bit-vector formulas and the field is still under active development. Despite this, only a few solvers support quantified bit-vector formulas: namely Boolector [PNB17], CVC4 [Nie+18b], Q3B [JS16], Yices [Dut15], and Z3 [WHM13].

This thesis contributes both to the theory and to the practice of solving satisfiability of quantified bit-vector formulas. As the theory is concerned, the thesis provides the precise complexity of this problem given that bit-widths and all numbers specified the formula are encoded in binary or decimal notation, which is often the case in practice. The thesis thus answers the open problem raised by Kovásznai et al. [KFB16]. We solve this problem by showing that the satisfiability problem of quantified bit-vector formulas with binary-encoded bit-widths is polynomially equivalent to the problem of solving satisfiability of second-order propositional formulas. This problem is known to be complete for the class of problems that can be solved by an alternating Turing machine that can use exponential time but only polynomially many alternations. The thesis also introduces novel formula simplifications, which are based on the known simplifications that leverage variables that occur only once in the input formula. Furthermore, the thesis presents a novel approach to solving quantified bit-vector formulas that is based on binary decision diagrams. This approach is further extended by approximations and abstractions that can improve its efficiency.

From the practical point of view, the introduced techniques have been implemented in an SMT solver Q3B and experimentally shown to outperform other state-of-the-art SMT solvers for quantified bit-vectors. Furthermore, the thesis offers experimental insight into the behaviour of quantified bit-vector formulas when the bit-widths of their variables are modified. The performed experiments confirm the natural hypothesis that satisfiability of the vast number of the quantified bit-vector formulas mostly does not depend on the bit-widths of the used variables. The thesis also introduces a technique that leverages this insight to speed-up quantified bit-vector solvers by first solving the input formula with smaller bit-width of variables and then verifying the obtained model or countermodel against the original formula.

There are three high-level topics that are recurring throughout the thesis.

- The first of these is the influence of bit-widths of variables of the formula on its satisfiability and the time that is necessary to solve the formula. This topic is present in the first part of the thesis that describes the precise complexity of deciding satisfiability of quantified bit-vector formulas with binary encoded bit-widths. It is also present in our experimental evaluation of the effect of reducing bit-widths of the formula's variables on its satisfiability. Furthermore, the topic of bit-widths is present in the approach of underapproximations and overapproximations, which are computed by reducing bit-widths of some of the variables, and in the approach of mixed approximations, which are computed by reducing bit-widths of all of the variables in the formula.

- The second recurring topic, which is related to the first one, is the usage of approximated formulas or abstract computation to improve the solving time of SMT solvers for quantified bit-vector formulas. This topic is present in the mentioned approximation approaches and in the approach that computes abstract and imprecise binary decision diagrams (BDDS) for the input formula, which represent only several bits of the results of the arithmetic operations.

- The third topic concerns binary decision diagrams. These are present in our approach to solving quantified bit-vector formulas and in the mentioned abstraction of results of bit-vector operations, which is described and implemented in terms of BDDS.

1.1 STRUCTURE OF THE THESIS

The thesis is structured as follows. Chapter 2 defines all the necessary concepts and introduces the notation that is used throughout the thesis. Chapter 3 presents current state of the art in the area of satisfiability modulo the theory of bit-vectors focused on the formulas with quantifiers.

The rest of the thesis exposes the topics that were described in the previous section. Namely, Chapter 4 investigates the computational complexity of satisfiability of quantified bit-vector formulas in which the values of bit-widths are encoded in binary or decimal notation.

The following two Chapters 5 and 6 introduce an approach to deciding satisfiability of quantified bit-vector formulas by using binary decision diagrams and abstractions of bit-vector operations. Chapter 7 extends known simplifications of formulas that contain unconstrained variables, i.e., variables that occur only once in the formula. The following Chapter 8 builds on the preceding three chapters and describes the implementation of our state-of-the-art SMT solver called Q3B, which uses the techniques introduced in these three chapters. Chapter 9 is devoted to experimental evaluation of Q3B: it provides the comparison with other state-of-the-art solvers for quantified bit-vector formulas and it also experimentally evaluates the effect of all the mentioned techniques on the performance of Q3B.

The final two Chapters 10 and 11 investigate the dependence of satisfiability of quantified bit-vector formulas on the bit-widths of the used variables. The former of these two chapters experimentally shows that in most formulas coming from practical applications, the satisfiability of the formula remains the same even after reducing the bit-widths in the formula to very small values. The latter chapter presents our ongoing research that uses this observation. Namely, the chapter introduces an approach that can speed up quantified bit-vector SMT solvers by using reductions of bit-widths in the input formula and extensions of the obtained models and countermodels.

1.2 AUTHOR'S PUBLICATIONS AND CONTRIBUTION

1.2.1 *Publications Related to the Thesis*

Each of the Chapters 4 to 8 and 10 is based on the eponymous paper of which I am the main author. All the papers were published in a journal or at an international conference. In the respective order of the chapters, the corresponding papers are:

INFORMATION PROCESSING LETTERS 135 On the Complexity of the Quantified Bit-Vector Arithmetic with Binary Encoding (M. Jonáš, J. Strejček) [JS18c].

My contribution: *I have proven all the results and written most of the paper.* *90%*

SAT 2016 Solving Quantified Bit-Vector Formulas using Binary Decision Diagrams (M. Jonáš, J. Strejček) [JS16].

My contribution: *I have proposed most of the techniques used in the approach, written a larger part of the paper, implemented the tool, performed all the experiments, and analyzed their results.* *75%*

ICTAC 2018 Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers (M. Jonáš, J. Strejček) [JS18a].

My contribution: *I have proposed most of the techniques used in the approach, written most of the paper, implemented the tool, performed all the experiments, and analyzed their results.* *90%*

SAT 2017 On Simplification of Formulas with Unconstrained Variables and Quantifiers (M. Jonáš, J. Strejček) [JS17].

My contribution: *I have proposed most of the ideas used in the paper, written most of the paper, implemented the tool, performed all the experiments, and analyzed their results.* *85%*

CAV 2019 Q3B: An Efficient BDD-Based SMT Solver for Quantified Bit-Vectors (M. Jonáš, J. Strejček) [JS19].

My contribution: *I have written most of the paper, implemented the tool, performed all the experiments, analyzed their results, and prepared the entire attached artifact, which was used during Artifact Evaluation.* *90%*

LPAR 2018 Is Satisfiability of Quantified Bit-Vector Formulas Stable Under Bit-Width Changes? (M. Jonáš, J. Strejček) [JS18b].

My contribution: *I have implemented the tool, performed all the experiments, analyzed their results, and written most of the paper.* *95%*

Although the chapters are based on the corresponding papers, some of the material has been rewritten and some new material was added.

- In addition to INFORMATION PROCESSING LETTERS 135, Chapter 4 contains a precise description of alternating Turing machines, more details on alternating complexity classes, and discusses in detail the polynomiality of the proposed reduction.

- In contrast to SAT 2016, Chapter 5 contains detailed description of bit-width approximations and proofs of their correctness. Moreover, the material has been updated to reflect the current implementation of the approach in Q3B.

- In addition to SAT 2017, Chapter 7 describes simplifications using *goal unconstrained* terms, which were not present in the original paper.

Moreover, two chapters besides the obvious ones are not based on any currently published papers. First, Chapter 9 contains an experimental evaluation of Q3B, which was performed anew for the thesis. Although the experimental evaluation of Q3B was contained in most of the papers [JS16; JS17; JS18a; JS19], the new evaluation improves the completeness of the experiments and strengthens their connection with the rest of the thesis. Second, Chapter 11 is based on the research that is currently ongoing and the chapter presents its current state. We expect to finish the research after this thesis is submitted and turn the chapter into a conference paper.

1.2.2 *Other Publications*

I have also co-authored three more publications. All three of them describe tools that have competed in the Competition on Software Verification (SV-COMP) [Bey19]. However, the papers are unrelated to the thesis and my contribution in them is only minor: about 5% in each of them.

TACAS 2016 Symbiotic 3: New Slicer and Error-Witness Generation (Competition Contribution) (M. Chalupa, M. Jonáš, J. Slabý, J. Strejček, M. Vitovská) [Cha+16]

TACAS 2017 Symbiotic 4: Beyond Reachability (Competition Contribution) (M. Chalupa, M. Vitovská, M. Jonáš, J. Slabý, J. Strejček) [Cha+17]

TACAS 2017 Optimizing and Caching SMT Queries in SymDIVINE (Competition Contribution) (J. Mrázek, M. Jonáš, V. Štill, H. Lauko, J. Barnat) [Mrá+17]

2 PRELIMINARIES

This chapter introduces *many-sorted first-order logic*, together with the detailed description of the theory of fixed-size bit-vectors. Further, it introduces *binary decision diagrams*, together with their ability to represent sets of bit-vectors.

2.1 MATHEMATICAL CONVENTIONS

We denote the set of non-negative integers as $\mathbb{N}$ and the set of positive integers as $\mathbb{N}^+$. If a and b are integers, we denote as $[a, b]$ the set of all integers x such that $a \leq x \leq b$. If not stated otherwise, all functions are considered as *total*. For defining functions in-place, we sometimes use the standard *lambda notation*, e.g., $\lambda x \lambda y.\, 2x + y$ is a binary function that sums two times its first argument x and its second argument y. We also sometimes write functions with a finite domain extensionally, e.g., $\{1 \mapsto 2, 3 \mapsto 8\}$ is a function with the domain $\{1, 3\}$ and the range $\{2, 8\}$, which assigns 2 to 1 and 8 to 3. If convenient, we work with functions as with sets of pairs. For example, the union of functions $f\colon A \to B$ and $g\colon C \to D$ where $A \cap C = \emptyset$ is a function $f \cup g\colon (A \cup C) \to (B \cup D)$. If f is a function, and x and v are arbitrary values, we denote as $f[x \mapsto v]$ the function defined by $f[x \mapsto v](x) = v$ and $f[x \mapsto v](y) = f(y)$ for all $y \neq x$ in the domain of f. When convenient, we identify a nullary function with the sole element in its range, e.g., for a nullary function $c\colon () \to A$, we write only c for the element $c() \in A$.

For a set A and an $n \in \mathbb{N}$, the set of all vectors of n elements over the set A is denoted as A^n. Vectors from A^n are written as lists of their elements; for example (a, c, b, c) is a vector in $\{a, b, c\}^4$. For $v \in A^n$ and $0 \leq i < n$, we denote as v_i the element of v on the position i. For example, if $v = (a, c, b, c)$, then $v_0 = a$ and $v_1 = c$.

2.2 MANY-SORTED FIRST-ORDER LOGIC

In this section, we define syntax and semantics of the many-sorted first-order logic. The definitions used are based on the SMT-LIB standard [BFT17], Handbook of Satisfiability [Bar+09], and the chapter on the many-sorted logic from the corresponding chapter of H. B. Enderton's monography [End01].

2.2.1 *Syntax*

SIGNATURE Let $\mathcal{S}$ be a countable set of *sort symbols* that contains a distinguished sort symbol *Bool*. For each sort $\sigma \in \mathcal{S}$, we have a countably infinite set of variables $vars_\sigma = \{x_1^\sigma, x_2^\sigma, \ldots\}$ such that all such sets are pairwise disjoint.

We denote the union of all *vars*$_\sigma$ as *vars*. A *signature* Σ is a triple $(\Sigma^s, \Sigma^f, \tau)$ consisting of

- a set of sort symbols $\Sigma^s \subseteq \mathcal{S}$ that contains the sort *Bool*,
- a set Σ^f of function symbols,
- a sort mapping τ, which to each function symbol $f \in \Sigma^f$ assigns its sort $\tau(f) = (\sigma_1, \sigma_2, \dots, \sigma_n, \sigma)$ for some $n \geq 0$ and $\sigma, \sigma_1, \sigma_2, \dots, \sigma_n \in \Sigma^s$.

The number n in the previous definition is called the *arity* of the corresponding function symbol. Function symbols with arity 0 are called *constants*. In the rest of this section, fix an arbitrary signature $\Sigma = (\Sigma^s, \Sigma^f, \tau)$.

TERMS AND FORMULAS The sets of Σ-*terms of sort* σ are defined inductively, simultaneously for all $\sigma \in \Sigma^s$:

1. for all $\sigma \in \Sigma^s$, each $x_i^\sigma \in vars_\sigma$ is a Σ-term of the sort σ,
2. if $f \in \Sigma^f$ is a function symbol of sort $(\sigma_1, \sigma_2, \dots, \sigma_n, \sigma)$ and $t_1, t_2, \dots, t_n$ are Σ-terms of sorts $\sigma_1, \sigma_2, \dots, \sigma_n$, respectively, then $f(t_1, t_2, \dots, t_n)$ is a Σ-term of the sort σ,
3. $\top$ and $\bot$ are Σ-terms of the sort *Bool*,
4. if φ_1 and φ_2 are Σ-terms of the sort *Bool*, then also $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, and $\varphi_1 \vee \varphi_2$ are Σ-terms of the sort *Bool*,
5. if φ is a Σ-term of the sort *Bool* and $x_i^\sigma \in vars$, then $\forall x_i^\sigma\, \varphi$ and $\exists x_i^\sigma\, \varphi$ are also Σ-terms of sort *Bool*,
6. if φ is a Σ-term of the sort *Bool* and t_1, t_2 are Σ-terms of a sort σ, then also $\mathtt{ite}_\sigma(\varphi, t_1, t_2)$ is a term of the sort σ.

Note that in contrast to the standard treatment of the single-sorted first-order logic, which differentiates between Σ-*formulas* and Σ-*terms*, in the many-sorted case, both of these are Σ-terms, albeit of the different sorts. Therefore, to make the distinction explicit, we call all Σ-terms of the sort *Bool* Σ-*formulas* and denote them by the lowercase Greek letters φ, ψ, ρ, etc. Additionally, Σ-formulas that have no proper subterms of the sort *Bool* as called *atomic* Σ-*formulas*. We say that an occurrence of a Σ-formula ψ in the Σ-formula φ has the *positive polarity* if the occurrence is under an even number of negations and that it has the *negative polarity* otherwise.

If t is a Σ-term, we denote the set free variables that occur in t as *vars*(t).

FORMULA MANIPULATIONS If φ is a Σ-formula and t, s are Σ-terms of the same sort, we use $\varphi[t \leftarrow s]$ to denote the results of substituting every occurrence of t in φ by s. More generally, for Σ-terms t_1, s_1, t_2, s_2, …, t_n, s_n, where each two Σ-terms t_i and s_i have the same sort, we denote as $\varphi[t_1 \leftarrow s_1, t_2 \leftarrow s_2, \dots, t_n \leftarrow s_n]$ the result of simultaneous substitution of each occurrence of t_i in φ by s_i. In particular, if x is a variable, then $\varphi[x \leftarrow t]$ is the result of substituting the variable x by the Σ-term t.

2.2.2 *Semantics*

STRUCTURES A Σ-*structure* $\mathcal{M}$ is a pair $(\nu, (\_)^{\mathcal{M}})$, where

- ν is a function that assigns to each sort $\sigma \in \Sigma^s$ a non-empty set $\nu(\sigma)$, which is called its domain. The function has to assign disjoint sets $\nu(\sigma) \cap \nu(\sigma') = \emptyset$ to all sorts $\sigma \neq \sigma'$ and to the sort *Bool*, it has to assign the two-element set $\nu(Bool) = \{1, 0\}$;
- $(\_)^{\mathcal{M}}$ is a function that assigns to each function symbol $f \in \Sigma^f$ of a sort $\tau(f) = (\sigma_1, \sigma_2, \dots, \sigma_n, \sigma)$ a function $f^{\mathcal{M}} : \nu(\sigma_1) \times \nu(\sigma_2) \times \dots \times \nu(\sigma_n) \to \nu(\sigma)$.

The elements of $\nu(\sigma)$ for some sort $\sigma \neq Bool$ are called *objects* of the given Σ-structure.

EVALUATION Let $\mathcal{M} = (\nu, (\_)^{\mathcal{M}})$ be a Σ-structure and μ be a compatible mapping, which assigns a value $\mu(x_i^\sigma) \in \nu(\sigma)$ to each variable $x_i^\sigma \in vars, \sigma \in \Sigma^s$. We define an evaluation function $[\![\_]\!]_\mu^{\mathcal{M}}$, which assigns a value $[\![t]\!]_\mu^{\mathcal{M}} \in \nu(\sigma)$ to each Σ-term of a sort σ. This function is defined recursively on the structure of the Σ-term:

1. For variables $[\![x_i^\sigma]\!]_\mu^{\mathcal{M}} = \mu(x_i^\sigma)$.
2. For applications of function symbols,
$$[\![f(t_1, \dots, t_n)]\!]_\mu^{\mathcal{M}} = f^{\mathcal{M}}([\![t_1]\!]_\mu^{\mathcal{M}}, \dots, [\![t_n]\!]_\mu^{\mathcal{M}}).$$
3. For Boolean constants, $[\![\top]\!]_\mu^{\mathcal{M}} = 1$ and $[\![\bot]\!]_\mu^{\mathcal{M}} = 0$.
4. For Boolean connectives:
   a) $[\![\neg\varphi_1]\!]_\mu^{\mathcal{M}} = 1$ if $[\![\varphi_1]\!]_\mu^{\mathcal{M}} = 0$; it is 0 otherwise.
   b) $[\![\varphi_1 \wedge \varphi_2]\!]_\mu^{\mathcal{M}} = 1$ if $[\![\varphi_1]\!]_\mu^{\mathcal{M}} = 1$ and $[\![\varphi_2]\!]_\mu^{\mathcal{M}} = 1$; it is 0 otherwise.
   c) $[\![\varphi_1 \vee \varphi_2]\!]_\mu^{\mathcal{M}} = 1$ if $[\![\varphi_1]\!]_\mu^{\mathcal{M}} = 1$ or $[\![\varphi_2]\!]_\mu^{\mathcal{M}} = 1$; it is 0 otherwise.
5. For quantifiers:
   a) $[\![\forall x_i^\sigma\, \varphi]\!]_\mu^{\mathcal{M}} = 1$ if $[\![\varphi]\!]_{\mu[x_i^\sigma \mapsto v]}^{\mathcal{M}} = 1$ for all values $v \in \nu(\sigma)$; it is 0 otherwise.
   b) $[\![\exists x_i^\sigma\, \varphi]\!]_\mu^{\mathcal{M}} = 1$ if there is a value $v \in \nu(\sigma)$ such that $[\![\varphi]\!]_{\mu[x_i^\sigma \mapsto v]}^{\mathcal{M}} = 1$; it is 0 otherwise.
6. For if-then-else function, $[\![\mathtt{ite}_\sigma(\varphi, t_1, t_2)]\!]_\mu^{\mathcal{M}} = [\![t_1]\!]_\mu^{\mathcal{M}}$ if $[\![\varphi]\!]_\mu^{\mathcal{M}} = 1$; it is $[\![t_2]\!]_\mu^{\mathcal{M}}$ otherwise.

INTERPRETATIONS A Σ-*interpretation* is a pair $(\mathcal{M}, \mu)$ of a Σ-structure and a compatible mapping μ that to each variable assigns a value of the corresponding sort. A Σ-interpretation $(\mathcal{M}, \mu)$ is said to *satisfy* a Σ-formula φ if $[\![\varphi]\!]_\mu^{\mathcal{M}} = 1$. We denote this fact as $(\mathcal{M}, \mu) \vDash \varphi$ and call such a Σ-interpretation $(\mathcal{M}, \mu)$ a *model of the* Σ*-formula* φ.

THEORIES A *theory* is a pair $\mathcal{T} = (\Sigma, I)$, where Σ is a signature and I is a non-empty class of Σ-interpretations, which are called *$\mathcal{T}$-models* or *models of the theory $\mathcal{T}$*. A Σ-interpretation from I that is a model of a Σ-formula φ is called a *$\mathcal{T}$-model of the Σ-formula φ*. A Σ-formula is called *$\mathcal{T}$-satisfiable* or *satisfiable modulo $\mathcal{T}$* if it has a $\mathcal{T}$-model; it is called *$\mathcal{T}$-unsatisfiable* or *unsatisfiable modulo $\mathcal{T}$* otherwise.

Two Σ-formulas φ and ψ are called *$\mathcal{T}$-equivalent* if the equality $[\![\varphi]\!]^{\mathcal{M}}_{\mu} = [\![\psi]\!]^{\mathcal{M}}_{\mu}$ holds for each $\mathcal{T}$-model $(\mathcal{M}, \mu)$. Two Σ-formulas φ and ψ are called *$\mathcal{T}$-equisatisfiable* if they are both $\mathcal{T}$-satisfiable or both $\mathcal{T}$-unsatisfiable. If φ is a Σ-formula and Γ is a set of Σ-formulas, we say that Γ *entails φ in $\mathcal{T}$*, written $\Gamma \models_{\mathcal{T}} \varphi$, if every Σ-interpretation in I that satisfies all Σ-formulas in Γ also satisfies φ. If $\emptyset \models_{\mathcal{T}} \varphi$, the formula φ is called *$\mathcal{T}$-valid* or a *theory lemma*. Note that this means that φ is satisfied by all Σ-interpretations in I.

2.3 THEORY OF FIXED-SIZE BIT-VECTORS

This thesis is focused on one particular theory, the theory of *fixed-size bit-vectors* (or bit-vector theory for short). Intuitively, the objects of models of this theory are vectors of bits, i.e. words from $\{0,1\}^+$, and the functions correspond to bit-wise operations on the individual bits and to arithmetic operations on the binary numbers that are represented by these bits. Bit-vectors are divided according to their length, or *bit-width*; for each possible bit-width the theory contains the respective sort. We now define syntax and semantics of the theory of fixed-size bit-vectors precisely.

2.3.1 *Syntax*

We denote the sort of bit-vectors of the bit-width n as $[n]$. The signature of the bit-vector theory is then $\Sigma_{BV} = (\Sigma^s_{BV}, \Sigma^f_{BV}, \tau_{BV})$, where the set of sort symbols is

$$\Sigma^s_{BV} = \{Bool\} \cup \{[n] \mid n \in \mathbb{N}^+\},$$

the set of function symbols is $\Sigma^f_{BV} = F \cup P$ with the set F defined as

$$\begin{aligned}
F = & \bigcup_{n \in \mathbb{N}^+} \left\{0^{[n]}, 1^{[n]}, \dots, (2^n - 1)^{[n]}\right\} \cup \\
& \cup \bigcup_{n \in \mathbb{N}^+} \left\{-^{[n]}, +^{[n]}, \times^{[n]}, /^{[n]}_u, \%^{[n]}_u, /^{[n]}_s, \%^{[n]}_s\right\} \cup \\
& \cup \bigcup_{n \in \mathbb{N}^+} \left\{\sim^{[n]}, \&^{[n]}, |^{[n]}, \ll^{[n]}, \gg_u{}^{[n]}, \gg_s{}^{[n]}\right\} \cup \\
& \cup \left\{\mathtt{concat}^{[m,n]} \mid m, n \in \mathbb{N}^+\right\} \cup \\
& \cup \bigcup_{m,n \in \mathbb{N}^+} \left\{\mathtt{signExtend}^{[n]}_m, \mathtt{zeroExtend}^{[n]}_m\right\} \cup \\
& \cup \left\{\mathtt{extract}^{[n]}_{j,i} \mid n, i, j \in \mathbb{N}^+, i \leq j < n\right\},
\end{aligned}$$

Symbol f	Sort $\tau_{BV}(f)$	Intended meaning
$0^{[n]}, 1^{[n]}, \ldots$	$[n]$	natural number constants
$\sim^{[n]}$	$[n] \to [n]$	bit-wise negation
$\&^{[n]}, \mid^{[n]}$	$[n] \times [n] \to [n]$	bit-wise and, or
$\ll^{[n]}, \gg_u{}^{[n]}$	$[n] \times [n] \to [n]$	logical shift left and right
$\gg_s{}^{[n]}$	$[n] \times [n] \to [n]$	arithmetic shift right
$-^{[n]}$	$[n] \to [n]$	arithmetic negation
$+^{[n]}, \times^{[n]}$	$[n] \times [n] \to [n]$	addition, multiplication
$/_u^{[n]}, \%_u^{[n]}$	$[n] \times [n] \to [n]$	unsigned division, remainder
$/_s^{[n]}, \%_s^{[n]}$	$[n] \times [n] \to [n]$	signed division, remainder
$\texttt{concat}^{[m,n]}$	$[m] \times [n] \to [m+n]$	concatenation
$\texttt{zeroExtend}_m^{[n]}$	$[n] \to [m+n]$	zero extension
$\texttt{signExtend}_m^{[n]}$	$[n] \to [m+n]$	sign extension
$\texttt{extract}_{j,i}^{[n]}$	$[n] \to [j-i+1]$	extraction from i-th to j-th bit
$=^{[n]}$	$[n] \times [n] \to Bool$	equality
$\leq_u^{[n]}, <_u^{[n]}$	$[n] \times [n] \to Bool$	unsigned less or equal, less than
$\leq_s^{[n]}, <_s^{[n]}$	$[n] \times [n] \to Bool$	signed less or equal, less than

Table 2.1: Function and predicate symbols of the bit-vector logic.

the set P defined as

$$P = \bigcup_{n \in \mathbb{N}^+} \left\{ =^{[n]}, \leq_u^{[n]}, <_u^{[n]}, \leq_s^{[n]}, <_s^{[n]} \right\},$$

and the sort mapping τ_{BV} for each of these symbols is given by Table 2.1. This table also describes an intended meaning for each of the symbols. We call the symbols from F as *bit-vector function symbols* and the symbols from P as *bit-vector predicate symbols*. The numbers n, m, i, and j in the sort $[n]$ and in all the defined symbols are called *scalars*.

Note that most all of the symbols are defined for each individual bit-width: for example, the signature contains a function symbol $+^{[4]}$, which is used to add the numeric value of two bit-vectors of bit-width 4, and it also contains the analogous function symbol $+^{[5]}$ for the addition of bit-vectors of bit-width 5.

2.3.2 *Semantics*

As was stated in the introduction of this section, objects in the bit-vector theory are vectors of bits. In particular, the set of objects corresponding to the sort $[n]$ with $n > 0$ is $\nu_{BV}([n]) = \{0,1\}^n$ and the set of objects corresponding to the sort *Bool* has to be $\nu_{BV}(Bool) = \{0,1\}$. For a more succinct notation, we sometimes write bit-vectors only as sequences of zeroes and ones, e.g., instead of writing $b = (1, 1, 0, 1)$, we write $b = 1011$. Note that the least significant bit b_0 is on the right.

To formally define the interpretation of the function and predicate symbols, we need to define the standard function that assigns a natural number from the interval $[0, 2^n - 1]$ to each bit-vector of bit-width n. Intuitively, the function assigns to each bit-vector a number that it represents in the binary notation. Formally, we define the function: $nat_n : \{0,1\}^n \to [0, 2^n - 1]$ as

$$nat_n(b) = \sum_{0 \leq i < n} 2^i b_i.$$

The function nat_n is for each $n > 0$ obviously a bijection and thus has the unique inverse. We denote the inverse of the function nat_n as bv_n. For example, the result of $nat_4(1101) = 13$ and $bv_4(7) = 0111$. Similarly, we define the function int_n that assigns to each bit-vector of bit-width n the corresponding integer in the *two's complement* representation from $[-2^{n-1}, 2^{n-1} - 1]$. In this representation, the sign of the number is encoded in the bit b_{n-1} of the bit-vector b: the negative numbers are expressed by setting b_{n-1} to 1. If the encoded number is positive, the rest of the bits encodes the value of the represented number; if the encoded number is negative, the rest of the bits encodes its difference from the least possible representable negative value -2^{n-1}. Formally, the function $int_n : \{0,1\}^n \to [-2^{n-1}, 2^{n-1} - 1]$ is defined as

$$int_n(b) = -2^{n-1} b_{n-1} + nat_{n-1}((b_i)_{0 \leq i < n-1}).$$

For example,

$$\begin{aligned} int_4(0000) &= 0, & int_4(1111) &= -1, \\ int_4(0001) &= 1, & int_4(1110) &= -2, \\ int_4(0111) &= 7, & int_4(1000) &= -8. \end{aligned}$$

The function int_n is also a bijection for each $n > 0$; we denote its inverse as sbv_n. In the further text, we denote the bit-vector $bv_n(2^n - 1)$ as $unsignedMax^{[n]}$, the bit-vector $sbv_n(-2^{n-1})$ as $signedMin^{[n]}$, and the bit-vector $sbv_n(2^{n-1} - 1)$ as $signedMax^{[n]}$.

Using these functions, Table 2.2 defines the function $(\_)^{BV}$ that assigns a function to each bit-vector function and predicate symbol. We denote the resulting Σ-structure $(\nu_{BV}, (\_)^{BV})$ as $\mathcal{M}_{BV}$. The models of the bit-vector theory are all Σ_{BV}-interpretations in which the Σ_{BV}-structure is precisely $\mathcal{M}_{BV}$. Formally, the bit-vector theory is the pair

Note that the SMT-LIB standard defines results for division by 0 and its remainder. The standard mimics the behaviour of the hardware circuits.

$$\mathcal{T}_{BV} = \left(\Sigma_{BV}, \{(\mathcal{M}_{BV}, \mu) \mid (\mathcal{M}_{BV}, \mu) \text{ is a } \Sigma_{BV}\text{-interpretation}\}\right).$$

Intuitively, the meaning of all function and predicate symbols is fixed in all models of the bit-vector theory; the models differ only in the values assigned to the variables.

Symbol f	Semantics $f^{\mathcal{BV}}$
$c^{[n]}$	$bv_n(c)$
$\sim^{[n]}$	$\lambda x.\,(1 - x_i)_{0 \le i < n}$
$\&^{[n]}$	$\lambda x \lambda y.\,(\min(x_i, y_i))_{0 \le i < n}$
$\mid^{[n]}$	$\lambda x \lambda y.\,(\max(x_i, y_i))_{0 \le i < n}$
$\ll^{[n]}$	$\lambda x \lambda y.\,bv_n((nat_n(x) \cdot 2^{nat_n(y)}) \bmod 2^n)$
$\gg_u{}^{[n]}$	$\lambda x \lambda y.\,bv_n(\lfloor nat_n(x)/2^{nat_n(y)} \rfloor)$
$\gg_s{}^{[n]}$	$\lambda x \lambda y.\,sbv_n(\lfloor int_n(x)/2^{nat_n(y)} \rfloor)$
$-^{[n]}$	$\lambda x.\,bv_n((2^n - nat_n(x)) \bmod 2^n)$
$+^{[n]}$	$\lambda x \lambda y.\,bv_n((nat_n(x) + nat_n(y)) \bmod 2^n)$
$\times^{[n]}$	$\lambda x \lambda y.\,bv_n((nat_n(x) \cdot nat_n(y)) \bmod 2^n)$
$/_u^{[n]}$	$\lambda x \lambda y.$ if $nat_n(y) = 0$ then $(1)_{0 \le i < n}$ else $bv_n(\lfloor nat_n(x)/nat_n(y) \rfloor)$
$\%_u^{[n]}$	$\lambda x \lambda y.$ if $nat_n(y) = 0$ then x else $bv_n(nat_n(x) \bmod nat_n(y))$
$/_s^{[n]}$	$\lambda x \lambda y.$ if $int_n(y) = 0 \wedge int_n(x) \ge 0$ then $(1)_{0 \le i < n}$ else if $int_n(y) = 0 \wedge int_n(x) < 0$ then $sbv_n(1)$ else $sbv_n(\lfloor int_n(x)/int_n(y) \rfloor)$
$\%_s^{[n]}$	$\lambda x \lambda y.$ if $int_n(y) = 0$ then x else $sbv_n(int_n(x) \bmod int_n(y))$
$\texttt{concat}^{[m,n]}$	$\lambda x \lambda y.\,bv_{m+n}(2^n \cdot nat_m(x) + nat_n(y))$
$\texttt{zeroExtend}_m^{[n]}$	$\lambda x.\,bv_{m+n}(nat_n(x))$
$\texttt{signExtend}_m^{[n]}$	$\lambda x.\,sbv_{m+n}(int_n(x))$
$\texttt{extract}_{j,i}^{[n]}$	$\lambda x.\,(x_{i+k})_{0 \le k < j-i+1}$
$=^{[n]}$	$\lambda x \lambda y.$ if $nat_n(x) = nat_n(y)$ then 1 else 0
$\le_u^{[n]}$,	$\lambda x \lambda y.$ if $nat_n(x) \le nat_n(y)$ then 1 else 0
$<_u^{[n]}$	$\lambda x \lambda y.$ if $nat_n(x) < nat_n(y)$ then 1 else 0
$\le_s^{[n]}$	$\lambda x \lambda y.$ if $int_n(x) \le int_n(y)$ then 1 else 0
$<_s^{[n]}$	$\lambda x \lambda y.$ if $int_n(x) < int_n(y)$ then 1 else 0

Table 2.2: Semantics of function and predicate symbols of the bit-vector logic.

2.3.3 *Examples*

In this section, we show several examples of both syntax and semantics of the bit-vector theory. The set of terms in this theory includes for example terms

- ${x_1}^{[8]}$ of sort [8],
- ${x_1}^{[8]} +^{[8]} {x_2}^{[8]}$ of sort [8],
- $\mathtt{concat}^{[8,2]}({x_1}^{[8]}, {x_3}^{[2]})$ of sort [10],
- $\mathtt{extract}^{[8]}_{4,1}({x_1}^{[8]})$ of sort [4], and
- $\mathtt{ite}_{[8]}(({x_1}^{[8]} {\leq_u}^{[8]} {x_2}^{[8]}), {x_1}^{[8]}, {x_2}^{[8]})$ of sort [8].

Further, it contains also for example the following formulas, i.e., terms of sort *Bool*:

- ${x_1}^{[8]} \ {\leq_u}^{[8]} \ ({x_1}^{[8]} +^{[8]} {x_2}^{[8]})$,
- $\exists {x_1}^{[8]} \forall {x_2}^{[8]} (({x_1}^{[8]} \times^{[8]} {x_2}^{[8]}) =^{[8]} 0^{[8]})$.

Suppose that μ is an arbitrary assignment such that $\mu({x_1}^{[8]}) = 00000111$, $\mu({x_2}^{[8]}) = 00000001$, and $\mu({x_3}^{[2]}) = 01$. Then

- $[\![{x_1}^{[8]}]\!]^{\mathcal{M}_{BV}}_{\mu} = 00000111$,
- $[\![{x_1}^{[8]} +^{[8]} {x_2}^{[8]}]\!]^{\mathcal{M}_{BV}}_{\mu} = 00001000$,
- $[\![\mathtt{concat}^{[8,2]}({x_1}^{[8]}, {x_3}^{[2]})]\!]^{\mathcal{M}_{BV}}_{\mu} = 0000011101$,
- $[\![\mathtt{extract}^{[8]}_{4,1}({x_1}^{[8]})]\!]^{\mathcal{M}_{BV}}_{\mu} = 0011$,
- $[\![\mathtt{ite}_{[8]}(({x_1}^{[8]} {\leq_u}^{[8]} {x_2}^{[8]}), {x_1}^{[8]}, {x_2}^{[8]})]\!]^{\mathcal{M}_{BV}}_{\mu} = 00000001$,
- $[\![{x_1}^{[8]} \ {\leq_u}^{[8]} \ ({x_1}^{[8]} +^{[8]} {x_2}^{[8]})]\!]^{\mathcal{M}_{BV}}_{\mu} = 1$, and
- $[\![\exists {x_1}^{[8]} \forall {x_2}^{[8]} (({x_1}^{[8]} \times^{[8]} {x_2}^{[8]}) =^{[8]} 0^{[8]})]\!]^{\mathcal{M}_{BV}}_{\mu} = 1$.

Therefore, the formula ${x_1}^{[8]} {\leq_u}^{[8]} ({x_1}^{[8]} +^{[8]} {x_2}^{[8]})$ is $\mathcal{T}_{BV}$-satisfiable, since it is satisfied by the Σ_{BV}-interpretation $(\mathcal{M}_{BV}, \mu) \in \mathcal{T}_{BV}$. This formula is not $\mathcal{T}_{BV}$-valid, since it is not satisfied by an arbitrary Σ_{BV}-interpretation $(\mathcal{M}_{BV}, \mu') \in \mathcal{T}_{BV}$ in which $\mu'({x_1}^{[8]}) = 11111111$ and $\mu'({x_2}^{[8]}) = 00000001$, because

$$255 = nat_8(11111111) > (nat_8(11111111) + nat_8(00000001)) \bmod 256 = 0.$$

The formula $\exists {x_1}^{[8]} \forall {x_2}^{[8]} (({x_1}^{[8]} \times^{[8]} {x_2}^{[8]}) =^{[8]} 0^{[32]})$ is both $\mathcal{T}_{BV}$-satisfiable and $\mathcal{T}_{BV}$-valid, as its value does not depend on the function μ in the interpretation, because it does not contain any free variables.

2.3.4 *Theory of Fixed-Size Bit-Vectors with Uninterpreted Functions*

A theory of fixed-size bit-vectors with uninterpreted functions has the same objects as the bit-vector theory, interprets all the bit-vector function and predicate symbols exactly as the bit-vector theory, but may contain additional function symbols, whose interpretation may be arbitrary. I.e., a theory of *fixed-size bit-vectors with uninterpreted functions* is every theory

$$(\Sigma_{UBV}, I) = ((\Sigma^s_{BV}, \Sigma^f_{UBV}, \tau_{UBV}), I),$$

where $\Sigma^f_{BV} \subseteq \Sigma^f_{UBV}$, for each $f \in \Sigma^f_{BV}$ we have $\tau_{BV}(f) = \tau_{UBV}(f)$, and

$$\begin{aligned} I = \{(\mathcal{M}, \mu) \mid (\mathcal{M}, \mu) &\text{ is a } \Sigma_{UBV}\text{-interpretation and } \mathcal{M} \text{ agrees with } \mathcal{M}_{BV} \\ &\text{on each symbol from } \Sigma_{BV}\}. \end{aligned}$$

For example, let f be a function symbol of sort $[32] \times [32] \to [32]$. Then the formula

$$\left(x^{[32]} + f(x^{[32]}, y^{[32]}) = y^{[32]}\right) \;\wedge\; \left(f(x^{[32]}, y^{[32]}) >_u 0^{[32]}\right)$$

is satisfiable modulo the theory of bit-vectors with uninterpreted functions, because it is for example satisfied by each Σ_{UBV}-structure $(\mathcal{M}, \mu)$, in which $\mu(x^{[32]}) = bv_{32}(0)$, $\mu(y^{[32]}) = bv_{32}(1)$, and $f^{\mathcal{M}}(x, y) = bv_{32}(1)$.

On the other hand, the formula

$$f(x^{[32]}, y^{[32]}) >_u 0 \;\wedge\; \left(x^{[32]} = z^{[32]}\right) \;\wedge\; f(z^{[32]}, y^{[32]}) = 0$$

over the same signature is not satisfiable in this theory.

2.3.5 *Notational conventions*

Since the rest of the thesis is mostly concerned with the bit-vector theory, we drop the $\mathcal{T}_{BV}$- prefix from terms such as $\mathcal{T}_{BV}$-term, $\mathcal{T}_{BV}$-formula, $\mathcal{T}_{BV}$-model, $\mathcal{T}_{BV}$-satisfiability, $\mathcal{T}_{BV}$-validity, and use only *term*, *formula*, *model*, *satisfiability* and *validity* instead.

We also omit bit-widths from all variables, numerals, function symbols and predicate symbols, if their bit-width can be inferred from the context. If the bit-width is not given and cannot be inferred from the context, it is supposed to be arbitrary. Therefore, the 32-bit versions of the formulas from the preceding subsection could be written as

- ${x_1}^{[32]} \leq_u ({x_1}^{[32]} + {x_2}^{[32]})$,
- $\exists {x_1}^{[32]} \forall {x_2}^{[32]} (({x_1}^{[32]} \times {x_2}^{[32]}) = 0)$.

If convenient, we also write the bit-vector numerals in binary instead in decimal notation. E.g., we write 0101 instead of the terms $nat_4(0101)^{[4]}$ or $5^{[4]}$.

In the following text, we also use variable names $x, y, z, \ldots$ instead of ${x_1}^{[n]}$, ${x_2}^{[n]}$, ${x_3}^{[n]}$ etc. Since all interpretations $(\mathcal{M}_{BV}, \mu)$ of the bit-vector theory share the same structure $\mathcal{M}_{BV}$, we usually do not mention the structure explicitly and only write that a formula is satisfied by an assignment μ. Moreover, as the value of a term depends only on the variables that occur in it, we usually do not write the values for the other variables when defining an assignment of values to variables. For example, we write that the formula $\varphi \equiv x^{[8]} + y^{[8]} = 0^{[8]}$ is satisfied by an assignment $\mu = \{x^{[8]} \mapsto 00000000, y^{[8]} \mapsto 00000000\}$ and that $[\![\varphi]\!]_\mu = 1$. Note that each such partial assignment corresponds to at least one complete assignment because each sort is interpreted by a non-empty set.

Some of the function symbols also have a standard shorter syntax. For example, instead of writing $\mathtt{extract}_{j,i}(t)$ it is traditional to write $t[j:i]$. Similarly, instead of writing $\mathsf{concat}(t_1, t_2)$, we write only $t_1 \cdot t_2$.

We also use the following traditional shortcuts for additional logical connectives:

- $\varphi \to \psi \quad \equiv \quad (\neg\varphi \vee \psi)$,
- $\varphi \leftrightarrow \psi \quad \equiv \quad (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$.

It can be shown that each formula that uses these additional logical connectives can be converted to an equivalent formula that uses only $\wedge$, $\vee$, and $\neg$ in polynomial time [Har16]. However, this conversion is more complicated than mere substitution of $\to$ and $\leftrightarrow$ by their definitions because this could lead to an exponentially larger formula.

As is usual, we denote the formula $\neg(t_1 = t_2)$ as $t_1 \neq t_2$.

2.3.6 *Normal Forms*

This subsection briefly introduces three normal forms of first-order formulas that are used throughout the thesis. For more details, see for example the Harris's monography on computational logic [Har09].

NEGATION NORMAL FORM A formula φ is in the negation normal form (NNF) if all negations in the formula are applied to atomic subformulas and for each subformula of form $\mathtt{ite}(\psi, t_1, t_2)$, the formula ψ is an atomic formula.

The reason for the second restriction is that `ite` *can by used to implement* $\neg\varphi$ *as* $\mathtt{ite}(\varphi, \bot, \top)$.

Each formula can be converted to an equivalent formula in NNF in polynomial time. First, consider each occurence of a subformula $\mathtt{ite}(\psi, t_1, t_2)$, where ψ is not an atomic formula and the smallest subformula ρ that contains the subterm $\mathtt{ite}(\psi, t_1, t_2)$. The subformula ρ can be rewritten to

$$\exists p\,(\rho[\psi \leftarrow p] \ \wedge \ (p \leftrightarrow \psi)),$$

where p is a fresh variable of sort *Bool*. All the introduced bi-implications can be removed in polynomial time [Har16]. Finally, all negations can be pushed towards atomic subformulas using De Morgan laws in linear time.

PRENEX NORMAL FORM A formula φ is in the prenex normal form (PNF) if it has form $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n\,(\psi)$, where $Q_i \in \{\exists, \forall\}$ for all $1 \leq i \leq n$ and ψ does not contain quantifiers.

Each formula can also be converted to an equivalent formula in the PNF in polynomial time. First, the formula is converted to the NNF and all its quantified variable names are in polynomial time renamed to be unique, if necessary. The quantifiers can then be pushed to the beginning of the formula in linear time by using the transformations

$$(\exists x\,(\rho_1)) \vee \rho_2 \;\rightsquigarrow\; \exists x\,(\rho_1 \vee \rho_2), \qquad (\exists x\,(\rho_1)) \wedge \rho_2 \;\rightsquigarrow\; \exists x\,(\rho_1 \wedge \rho_2)$$

and the analogous ones for the universal quantifier.

SKOLEM NORMAL FORM A formula φ is in the Skolem normal form (SNF) if it has form $\forall x_1 \forall x_2 \ldots \forall x_n\,(\psi)$, where ψ does not contain quantifiers.

Each formula can be converted to an *equsatisfiable* formula in the SNF in polynomial time by first converting the formula to PNF and by performing *Skolemization* on the result. Skolemization is a process that for each formula ρ produces an equisatisfiable formula *skolemize*(ρ). The formula *skolemize*(ρ) is obtained by first replacing each occurrence of each existentially quantified variable x in ρ by a fresh function symbol f_x, whose arguments are all variables that are universally quantified before x, and then removing the existential quantifier for x.

Because of the new function symbols, the result is in general not equivalent to the original formula.

2.4 BINARY DECISION DIAGRAMS

A binary decision diagram (BDD) is a data structure that can succinctly represent Boolean functions, i.e., functions from $\{0,1\}^n$ to $\{0,1\}$ for some $n \in \mathbb{N}^+$. Formally, a BDD is a rooted binary directed acyclic graph that has at most two leaves, labelled by 0 and 1, and inner nodes labelled by the names of the formal arguments of the represented Boolean function. Each inner node has two children, called *a high child* and *a low child*, which intuitively represent the result of setting the corresponding formal argument of the function to 1 and 0, respectively. For example, Figure 2.1 shows a BDD b that represents a binary function $f(x,y) = (x \text{ xor } y)$. According to the traditional notation, the high children are marked by solid edges, the low children are marked by dotted edges. Given a BDD that represents a Boolean function f and an assignment of values to the arguments of the function, the value of f can be computed by traversing the BDD as follows: start in the root node; if the value of the argument corresponding to the current node is 1, continue to the high child, otherwise continue to the low child; continue with the traversal until reaching a leaf node and return its label. Given a BDD b and an assignment μ, we denote the result of the function represented by b as $[\![b]\!]_\mu$. For example, the result of evaluating BDD b from Figure 2.1 under an assignment $\mu = \{x \mapsto 0, y \mapsto 1\}$ is $[\![b]\!]_\mu = 1$. The trivial BDDs $\boxed{0}$ and $\boxed{1}$ represent constant functions *false* (0) and *true* (1), respectively.

Alternatively, binary decision diagrams can be used to represent a set of satisfying assignments (also called *models*) of a Boolean formula φ. Such a BDD represents a function that has Boolean variables of the formula φ as formal arguments and that evaluates to 1 in a given assignment iff the assignment is

In other words, the BDD represents the characteristic function of the set of models.

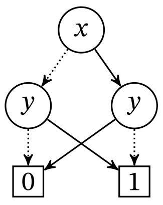
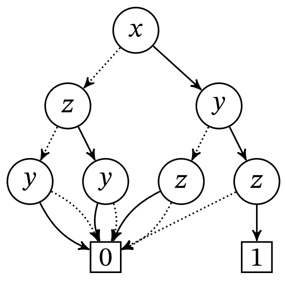
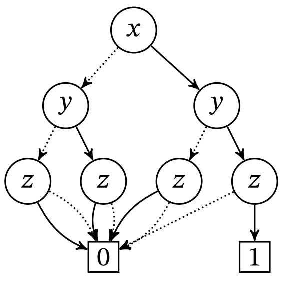


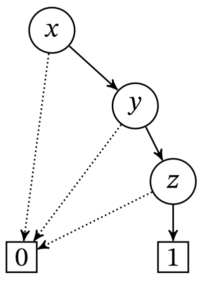
Figure 2.1: BDD for (x xor y)



(a) Non-ordered BDD.



(b) Ordered BDD.



(c) Reduced and ordered BDD

Figure 2.2: BDDs for the formula $(x \wedge y \wedge z)$.

a model of the formula φ. For example, the BDD of Figure 2.1 also represents the set of assignments satisfying the Boolean formula $(x \wedge \neg y) \vee (\neg x \wedge y)$.

In the rest of the thesis, we suppose that all binary decision diagrams are *reduced* and *ordered*. A BDD is *ordered* if, for all pairs of paths in the BDD, the order of common variables is the same. A BDD is *reduced* if it does not contain multiple copies of an identical induced subgraph and does not contain any inner node with the same high and low child. For example, the BDD in Figure 2.2a is not ordered, because the order of variables on the leftmost path is x, z, y and on the rightmost path it is x, y, z. The BDD in Figure 2.2b is ordered, but it is not reduced, because for example the leftmost node labelled by z has the same low and the high child, i.e., the node $\boxed{0}$. This is also the case for two other nodes labelled by z. Therefore, all these nodes can be removed and their incoming edges can be rerouted to their successor. The resulting BDD would still not be reduced, as the leftmost node labelled by y would have the node $\boxed{0}$ as both the low and the high child. Its removal yields the reduced and ordered BDD, which is shown in the Figure 2.2c. It has been shown that reduced and ordered BDDs are canonical – given a variable order, there is exactly one reduced and ordered BDD for each given function [Bry86].

Binary decision diagrams can be also used to represent an arbitrary *bit-vector function*, i.e., a function of type $\{0,1\}^n \to \{0,1\}^k$ that assigns a bit-vector value to each assignment of Boolean variables. Such a function of a bit-width k (i.e., the produced bit-vectors have the bit-width k) can be represented by a vector $\overline{b} = (b_i)_{0 \leq i < k}$ of k BDDs. The result of this function for a given assignment μ is then the bit-vector $[\![\overline{b}]\!]_\mu = ([\![b_i]\!]_\mu)_{0 \leq i < k}$. For example, Figure 2.3 shows a vector of BDDs representing addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of

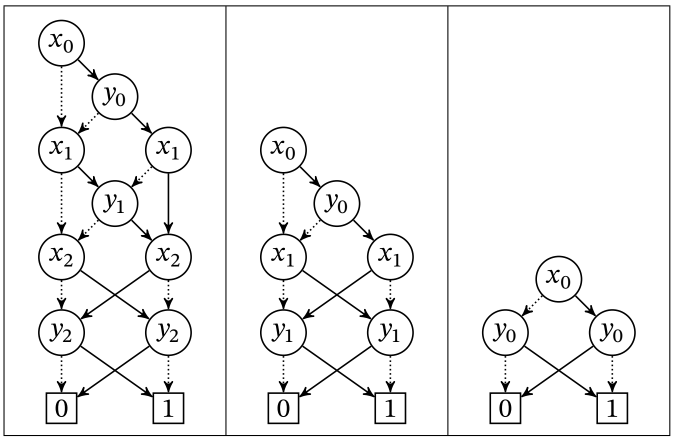


Figure 2.3: Vector of BDDs representing the addition $x_2x_1x_0+y_2y_1y_0$ of two bit-vectors of size 3. The least-significant bit of the result is on the right.

size 3, where each of the bit-vectors is represented by three Boolean variables. For example, for the assignment

$$\{x_2 \mapsto 0, x_1 \mapsto 0, x_0 \mapsto 1, y_2 \mapsto 0, y_1 \mapsto 0, y_0 \mapsto 1\},$$

the result $[\![\overline{b}]\!]_\mu$ is the bit-vector 010. In the following text, we denote the set of all BDDs as `BDD` and the set of all vectors of BDDs as `BDDvec`. We use the overlined symbols for both vectors of BDDs and bit-vectors.

2.4.1 *Operations on Binary Decision Diagrams*

It has been shown by Bryant [Bry86] that given a pair of BDDs for Boolean functions f and g, one can compute a BDD for an arbitrary Boolean connective by a recursive procedure called `Apply` in polynomial time. In particular, it is possible to compute BDDs for the functions $\lambda\overline{x}.\,f(\overline{x}) \wedge g(\overline{x})$ and $\lambda\overline{x}.\,f(\overline{x}) \vee g(\overline{x})$. A BDD for negation can be obtained by exchanging leaf nodes 0 and 1. We denote the functions for computing *conjunction*, *disjunction*, and *negation* of BDDs by the infix notations &, |, and !, respectively. Using these operations, a BDD for an arbitrary Boolean formula can be constructed by computing the corresponding BDDs for all subformulas recursively from the smallest ones. The procedure `Apply` can be also used to easily compute functions representing *equivalence* and *exclusive or* of two BDDs with the infix notations $\leftrightarrow$ and xor, respectively. We also define the *if-then-else* function that gets three BDDs a, b, c and returns the BDD `ite`(a, b, c) equivalent to $(a \,\&\, b) \mid (!a \,\&\, c)$.

Bryant has also described a recursive function `Restrict`, which modifies a given BDD by setting selected variables to given values. Using this function and the functions for logical connectives, it is possible to eliminate variable x from a given BDD representing the function $\lambda x \lambda\overline{y}.\,f(x, \overline{y})$ existentially or universally by computing the BDDs for $\lambda\overline{y}.\,f(0, \overline{y}) \vee f(1, \overline{y})$ or $\lambda\overline{y}.\,f(0, \overline{y}) \wedge f(1, \overline{y})$, respectively. Therefore, it is possible to compute a BDD corresponding to an arbitrary quantified Boolean formula.

```
bvec_add($\overline{a}$, $\overline{b}$)
{
  $\overline{result}$ ← ($\boxed{0}, \boxed{0}, \dots, \boxed{0}$) with the bit-width $k$;
  $carry$ ← $\boxed{0}$;
  for $i$ from 0 to $k$ - 1 {
    $result_i$ ← $a_i$ xor $b_i$ xor $carry$;
    $carry$ ← $(a_i$ & $b_i) \mid (carry$ & $(a_i \mid b_i))$;
  }
  return $\overline{result}$;
}

bvec_mul($\overline{a}$, $\overline{b}$)
{
  $\overline{result}$ ← ($\boxed{0}, \boxed{0}, \dots, \boxed{0}$) with the bit-width $k$;
  for $i$ from 0 to $k$ - 1 {
    $\overline{added}$ ← bvec_add($\overline{result}, \overline{a}$);
    for $j$ from $i$ to $k-1$ {
      $result_j$ ← ite($b_i, added_j, result_j$);
    }
    $\overline{a}$ ← bvec_shl($\overline{a}$, 1);
  }
  return $\overline{result}$;
}
```

Listing 2.1: Functions bvec_add and bvec_mul implementing *addition* (+) and *multiplication* ($\cdot$) on vectors $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$ of BDDs.

Further, given two vectors of BDDs that represent bit-vector functions f and g of the same bit-width, a vector of BDDs for the function $\lambda\overline{x}.\, f(\overline{x}) + g(\overline{x})$, where + is the *arithmetic addition* of two bit-vectors representing binary numbers, can also be computed by using the basic logical operations on BDDs representing the individual bits. Listing 2.1 shows an algorithm for this computation, which is implemented for example in the BDD package BuDDy[1]. The listing also shows the algorithm for the *arithmetic multiplication*. This algorithm takes as the input two vectors $\overline{a}$ and $\overline{b}$ of BDDs. Intuitively, the algorithm starts with the result set to 0 and, for each bit b_i of the second argument $\overline{b}$, the result stays the same if b_i is 0 and $2^i \cdot \overline{a}$ is added to the result if b_i is 1. Note that the inner `for`-cycle responsible for this addition ignores the i least significant bits as $2^i \cdot \overline{a}$ has 0 in these bits. The computation of $2^i \cdot \overline{a}$ is performed by iteratively shifting $\overline{a}$ left by one bit using the function bvec_shl, which shifts the vector of BDDs in its first argument by the number of bits given by the second argument. The remaining functions such as *unsigned division* or *unsigned remainder* can also be computed in a similar way.

Finally, given two vectors of BDDs that represent bit-vector functions f and g of the same bit-width, it is also possible to compute the BDD for the Boolean function that represents their *equality* $\lambda\overline{x}.\, f(\overline{x}) = g(\overline{x})$, the BDD for their *unsigned inequality* $\lambda\overline{x}.\, f(\overline{x}) \leq_u g(\overline{x})$, and the BDD for their *signed inequality*

1 http://sourceforge.net/projects/buddy

```
bvec_eq($\overline{a}$, $\overline{b}$)
{
  $result$ ← $\boxed{1}$;
  for $i$ from 0 to $k$ - 1 {
    $result$ ← $result$ & $(a_i \leftrightarrow b_i)$;
  }
  return $result$;
}

bvec_leq($\overline{a}$, $\overline{b}$)
{
  $result$ ← $\boxed{1}$;
  for $i$ from 0 to $k$ - 1 {
    $result$ ← $(!a_i$ & $b_i) \mid (result$ & $(a_i \leftrightarrow b_i))$
  }
  return $result$;
}
```

Listing 2.2: Functions bvec_eq and bvec_leq implementing *equality* (=) and *unsigned inequality* ($\leq_u$) of vectors $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$ of BDDs.

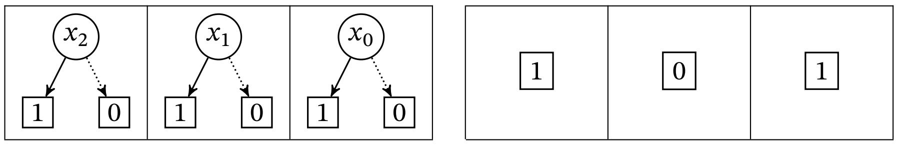


Figure 2.4: Vectors of BDDs representing 3-bit variable $x_2 x_1 x_0$ (left) and numeral $5^{[3]} = 101$ (right).

$\lambda \overline{x}. f(\overline{x}) \leq_s g(\overline{x})$. Listing 2.2 shows algorithms for *equality* and *unsigned inequality*, which again correspond to the implementation in BuDDy. The algorithm for *signed inequality* is similar

Starting with the trivial construction of vectors of BDDs for variables and numerals shown on Figure 2.4 and applying the algorithms mentioned above, one can easily implement the function t2BDDvec, which converts a bit-vector term of the sort $[n]$ to the vector of n BDDs representing the function computed by the term. Consequently, it is possible to define a function f2BDD, which converts a bit-vector formula to the corresponding BDD representing all assignments satisfying the formula. The function f2BDD hence provides a straightforward satisfiability check: a formula ψ is satisfiable if and only if the result of f2BDD(ψ) is a BDD where the leaf 1 is reachable from the root. For ordered and reduced BDDs, this is always the case unless the BDD is $\boxed{0}$.

3 QUANTIFIED BIT-VECTOR SATISFIABILITY: STATE OF THE ART

This chapter summarizes the state of the art in solving satisfiability of quantified bit-vector formulas.

All the approaches for deciding satisfiability of quantified bit-vector formulas build on the ability to decide satisfiability of *quantifier-free* bit-vector formulas. This is traditionally done by converting the input bit-vector formula either eagerly or lazily [Had+14] to the equisatisfiable propositional formula by a process called *bit-blasting* [KS08] and solving it by an off-the-shelf SAT solver. During bit-blasting, a propositional variable is introduced for each bit of each bit-vector variable in the input formula and the bit-vector operations are modeled using propositional connectives in a way that mimics the hardware circuits for these operations. We do not explain the solving of *quantifier-free* formulas in more detail and instead turn to *quantified* formulas, which are the main topic of the thesis.

3.1 MODEL-BASED QUANTIFIER INSTANTIATION

Solving of quantified bit-vector formulas was first supported by the solver Z3 in 2013 [WHM13] and for a limited set of *exists/forall* formulas with only a single quantifier alternation by Yices in 2015 [Dut15]. Both of these solvers decide quantified formulas by *quantifier instantiation*, in which the universally quantified variables in the Skolemized input formula are repeatedly instantiated by quantifier-free terms until the resulting quantifier-free formula is either unsatisfiable or a model of the original formula is found. Note that for general formulas, the solver for quantifier-free formulas has to be able to handle uninterpreted symbols, since Skolemization introduces new uninterpreted function symbols if the formula contains at least one existential quantifier in scope of a universal quantifier. To obtain terms that represent values of Skolem functions in the model, Z3 uses template-based model finding, where several templates for these functions are provided. For example, these may include templates for linear functions, affine functions, etc.

After performing Skolemization, the input formula is of the form

$$\varphi \land \forall x_1 \forall x_2 \ldots \forall x_n \, (\psi),$$

where φ and ψ are quantifier-free formulas that may contain uninterpreted function symbols. First, the solver for quantifier-free formulas is invoked to check the satisfiability of the formula φ. If φ is unsatisfiable, then the entire formula is unsatisfiable. If φ is satisfiable, the solver for quantifier-free formulas returns its model $\mathcal{M}$ and another call to the solver is made to determine whether $\mathcal{M}$ is also a model of $\forall x_1 \forall x_2 \ldots \forall x_n \, (\psi)$. This is achieved by asking the solver whether the quantifier-free formula $\neg\widehat{\psi}$ is satisfiable, where $\widehat{\psi}$ is the formula ψ with free variables and uninterpreted function symbols replaced by their corresponding values in $\mathcal{M}$. If $\neg\widehat{\psi}$ is not satisfiable, then the structure $\mathcal{M}$ is indeed a model of the formula $\forall x_1 \forall x_2 \ldots \forall x_n\,(\psi)$, therefore the entire formula is satisfiable and $\mathcal{M}$ is its model. If $\neg\widehat{\psi}$ is satisfiable, we get bit-vector values $v_1, \ldots, v_n$ such that $\neg\widehat{\psi}[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$ holds. To rule out $\mathcal{M}$ as a model, the instance $\psi[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$ of the quantified formula is added to the quantifier-free part of the formula; i.e. the formula φ is modified to

$$\varphi' \;\equiv\; \varphi \wedge \psi[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n],$$

and the procedure is repeated.

Example 3.1. *Consider the formula $3 <_u x \wedge \forall y\,(x \neq 2 \times y)$, where all variables have bit-width 32. The subformula $3 <_u x$ is satisfiable and $\{x \mapsto bv_{32}(4)\}$ is its model. However, it is not a model of the formula $\forall x\,(x \neq 2 \times y)$, since the solver for quantifier-free formulas called on the formula $4 \neq 2 \times y$ returns $\{y \mapsto bv_{32}(2)\}$ as a model. The next step is to decide the satisfiability of the formula $3 <_u x \wedge (x \neq 2 \times 2)$. This formula is satisfiable and $\{x \mapsto bv_{32}(5)\}$ is its model. Moreover, it is also a model of $\forall y\,(x \neq 2 \times y)$ as the formula $5 = 2 \times y$ is unsatisfiable. Hence, the input formula is satisfiable and $\{x \mapsto bv_{32}(5)\}$ is its model.*

This algorithm is trivially terminating, since there is only a finite number of distinct models $\mathcal{M}$ of φ. However, in some cases, exponentially many such models have to be ruled out before the solver is able to find a correct model or decide unsatisfiability of the whole formula. To overcome this issue, Z3 does not use only instances of the form $\psi[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$, but employs heuristics such as E-matching [DNS05; MB07] or symbolic quantifier instantiation [WHM13], which try to instantiate variables $x_1, \ldots, x_n$ not only by bit-vector values but also by more complex terms that can potentially rule out more spurious models and thus significantly reduce the number of iterations of the algorithm. In practice, suitable ground terms substituted for quantified variables are selected only from subterms of the input formula. This strategy unfortunately brings also some drawbacks. For example, the formula

$$(x = 16 \times y \;+\; 16 \times z) \;\wedge\; \forall v\,(x \neq 16 \times v)$$

with all variables of bit-width 32 is unsatisfiable as the subformula $\forall v\,(x \neq 16 \times v)$ is satisfied precisely for values of x that are not multiples of 16, while $x = 16 \times y + 16 \times z$ implies that x is a multiple of 16. The general quantifier instantiation could prove the unsatisfiability of this formula easily by using the instance $\psi[v \leftarrow y + z]$ of $\psi \equiv (x \neq 16 \times v)$. However, neither Z3 nor Yices consider this instance as $y + z$ is not a subterm of the formula. As the result, these tools cannot decide satisfiability of this formula within a reasonable time limits.

3.2 COUNTER-EXAMPLE GUIDED MODEL SYNTHESIS

Next solver that was able to solve quantified bit-vector formulas was Q3B in 2016. However, the approach of Q3B is discussed in detail in later chapters of this thesis and we therefore do not describe it here.

After Q3B, the following solver that was able to solve quantified bit-vector formulas was Boolector in 2017, using also an approach based on quantifier instantiation [PNB17]. Unlike Z3, in which the terms representing Skolem functions are obtained based on predefined templates, Boolector uses a counterexample-guided model synthesis (CEGMS) approach, in which the terms representing Skolem functions are synthesized from the predefined grammar. The synthesis of the suitable term from this grammar that satisfies given conditions is then implemented by *enumerative learning* [Udu+13]. Generally, the synthesis of the term can be implemented by using any syntax-guided synthesis approach [Alu+15].

Moreover, Boolector also employs a *dual solver*, which tries to prove that the input formula is unsatisfiable by proving that its negation with all added implicit existential quantifiers is satisfiable. Effectively, the dual solver tries to synthesize terms whose substitution for *universally* quantified variables in the formula makes it unsatisfiable. In Boolector, the standard and dual solver are run in parallel. The following example illustrates the dual solver.

Example 3.2. *Consider the formula*

$$(x = 16 \times y \;+\; 16 \times z) \;\land\; \forall v\,(x \neq 16 \times v)$$

from the previous section. After adding all implicit existential quantifiers, negating the formula and converting it to the Skolem normal form, one obtains the formula

$$\varphi_n \;\equiv\; \forall x \forall y \forall z\,((x \neq 16 \times y \;+\; 16 \times z) \;\lor\; (x = 16 \times f_v(x, y, z))).$$

The CEGMS *performs quantifier instantiation and after few iterations comes up with a potential model, which to the Skolem function $f_v(x, y, z)$ assigns the bit-vector term $y+z$. Similarly as in the previous section, this model is then verified by substituting it to the formula φ_n and using a solver for quantifier-free bit-vectors on the negation of the resulting formula. The formula φ_n is satisfiable, and the original formula φ is hence unsatisfiable.*

3.3 QUANTIFIER INSTANTIATION BASED ON INVERTIBILITY CONDITIONS

More recently, in 2018, support of quantified bit-vector formulas has also been implemented into the SMT solver CVC4 [Nie+18b]. The approach of CVC4 is also based on quantifier instantiation, but instead of synthesizing terms from the defined grammar as Boolector, CVC4 uses predetermined rules based on *invertibility conditions*, which directly give terms that may prune *all* spurious models without using the potentially expensive counterexample-guided synthesis.

Invertibility conditions give the sufficient and necessary conditions under which a formula has a solution for a given variable x. For example, given arbitrary terms t, s of bit-width n that do not contain t:

- The invertibility condition for x in $x + t = s$ is $\top$, because x can always be set to the value of $s - t$.
- The invertibility condition for x in $x \times s \neq t$ is $s \neq 0 \vee t \neq 0$, because unless $s = 0$, the expression $x \times s$ can be evaluated to at least two values, one of which must be distinct from t; if on the other hand $s = 0$, the formula has a solution if and only if $t \neq 0$.
- The invertibility condition for x in $x \times s = t$ is $(-s \mid s) \& t = t$, because for each bit-vector s, the result of $-s \mid s$ has as many least significant 0 bits as s and all its other bits set to 1. The condition $(-s \mid s) \,\&\, t = t$ therefore holds if and only if the result of t has at least as many least significant zeroes as the result of s.

Example 3.3. *Consider the formula $3 <_u x \wedge \forall y\,(x \neq 2 \times y)$ from Example 3.1 and the moment in this example where $\varphi' \equiv 3 <_u x$ and its model $\{x \mapsto bv_{32}(4)\}$ was identified not to be a model of $\forall y\,(x \neq 2 \times y)$ because the assignment $\{y \mapsto bv_{32}(2)\}$ satisfies the formula $4 = 2 \times y$. Instead of adding the precise instance $x \neq 2 \times 2$, which prohibits only the value $x = bv_{32}(4)$ as in the model-based quantifier instantiation, the quantifier instantiation based on invertibility conditions prohibits all values of x with the same problem.*

Observe that the subformula $\forall y\,(x \neq 2 \times y)$ prohibits all the values of x for which there is y such that $x = 2 \times y$. Based on the invertibility condition for y, this is precisely if $(-2 \mid 2) \,\&\, x = x$, which is satisfied precisely for the even values of x. Therefore, the formula φ' is modified to

$$\varphi'' = 3 <_u x \wedge \neg((-2 \mid 2) \,\&\, x = x)$$

and in the following iteration, the actual model $\{x \mapsto bv_{32}(5)\}$ is found. Observe that thanks to invertibility conditions, all even values of x were prohibited in one step.

The general approach is more complicated than the previous example, because it needs to handle cases where the quantified variable x occurs multiple times in the input formula or even in its single atomic subformula. We do not describe the complete approach here, since it is not necessary for the rest of the thesis.

3.4 TIMELINE

To put the mentioned techniques in context, Figure 3.1 shows the timeline of the described techniques together with our techniques described in Chapters 5 and 6.

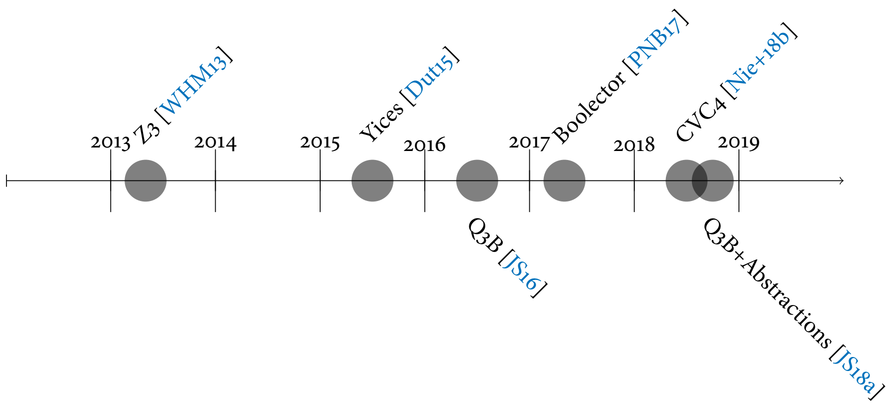


Figure 3.1: Timeline of SMT techniques for quantified bit-vectors with the corresponding solvers.

4 ON THE COMPLEXITY OF THE QUANTIFIED BIT-VECTOR ARITHMETIC WITH BINARY ENCODING

The computational complexity of the satisfiability problem for quantified bit-vector formulas has been shown to be **PSPACE**-complete and to become even **NEXPTIME**-complete when uninterpreted functions are allowed in addition to quantifiers [WHM13].

However, these results suppose that all scalars in the formula are represented in the unary encoding, which is not the case in practice, because in most of real-world applications, bit-widths and numerals are encoded logarithmically. For example, the format SMT-LIB [BFT17], which is an input format for most of the state-of-the-art SMT solvers, represents all scalar values as decimal numbers. Such representation can be exponentially more succinct than the representation using unary-encoded scalars. The satisfiability problem for bit-vector formulas with binary-encoded scalars has been recently investigated by Kovásznai et al. [KFB16]. They have shown that the satisfiability of quantified bit-vector formulas with binary-encoded scalars and with uninterpreted functions is complete for the class **2–NEXPTIME**. The situation for the same problem without uninterpreted functions was not so clear: deciding satisfiability of quantified bit-vector formulas with binary-encoded scalars and *without* uninterpreted functions (we denote this problem as BV2 satisfiability) had been known to be in **EXPSPACE** and to be **NEXPTIME**-hard, but its precise complexity had remained unknown [KFB16].

In this chapter, we present our solution of this open problem by identifying the complexity class for which BV2 satisfiability is complete. We use the notion of an alternating Turing machine introduced by Chandra et al. [CKS81] and show that the BV2 satisfiability problem is complete for the class **AEXP**(poly) of problems solvable by an alternating Turing machine using exponential time, but only a polynomial number of alternations.

4.1 ENCODINGS OF SCALARS

There are more ways to encode scalars occurring in the bit-vector formula: in the *unary encoding* or in a *logarithmic encoding*. For example, the bit-widths in the formula $x^{[64]} + y^{[64]}$ may be considered of size 64 if they are represented in unary, of size 7 if they are represented in binary as 1000000, or of size 2 if they are represented in decimal as 64. Note that the operation + may be regarded as having the size 1 because its sort is completely determined by the sorts of its arguments. In this chapter, we focus only on formulas using the *binary encoding*. This covers all logarithmic encodings including the decimal, since all of them are polynomially reducible to each other.

	Expression	Size
Numeral	$\lvert c^{[n]} \rvert$	$L(c) + L(n)$
Variable	$\lvert x^{[n]} \rvert$	$1 + L(n)$
Operation	$\lvert o(t_1, \dots, t_k, i_1, \dots, i_p) \rvert$	$1 + \sum_{1 \leq i \leq k} \lvert t_i \rvert + \sum_{1 \leq j \leq p} L(i_j)$
Quantifier	$\lvert Qx^{[n]} \varphi \rvert$	$\lvert x^{[n]} \rvert + \lvert \varphi \rvert$

Table 4.1: Recursive definition of the formula size. Operations include logical connectives, function symbols, and predicate symbols. Each t_i denotes a subterm or a subformula, each i_j denotes a scalar argument of an operation, and $Q \in \{\exists, \forall\}$ [KFB16].

In the binary encoding, $L(n)$ bits are needed to express the number n, where $L(0) = 1$ and $L(n) = \lfloor \log_2 n \rfloor + 1$ for all $n > 0$. The entire formula is encoded in the following way: each numeral $c^{[n]}$ has both its value c and bit-width n encoded in binary, each variable $x^{[n]}$ has its bit-width n encoded in binary, and all scalar arguments of functions, such as the bounds of the `extract` function, are encoded in binary. The size of the formula φ is denoted $|\varphi|$. The recursive definition of $|\varphi|$ is given in Table 4.1. For quantified formulas with binary-encoded scalars, we define the corresponding satisfiability problem:

Definition 4.1 (BV2 satisfiability problem [KFB16])**.** *The* BV2 satisfiability problem *is to decide satisfiability of a given closed quantified bit-vector formula with all scalars encoded in binary.*

4.2 ALTERNATION COMPLEXITY

In this section, we define the concept of an *alternating Turing machine* (ATM) introduced by Chandra, Kozen, and Stockmeyer [CKS81] and briefly recall properties of complexity classes defined by ATMs. We assume basic familiarity with *nondeterministic Turing machines* and basic concepts from the complexity theory, which can be found for example in Kozen [Koz06].

Definition 4.2 (Alternating Turing Machine)**.** *An* alternating Turing machine *M is an 8-tuple*

$$M = (Q, \Sigma, \Gamma, \sqcup, \rhd, \delta, q_0, type),$$

where

- *Q is a finite set of* states,
- *Σ is a finite* input alphabet,
- *Γ is a finite* tape alphabet *such that $\Sigma \subseteq \Gamma$,*
- *$\sqcup \in \Gamma \setminus \Sigma$ is the* blank symbol,

- $\rhd \in \Gamma \setminus \Sigma$ *is the* left endmarker,
- $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$ *is the* transition function,
- $q_0 \in Q$ *is the initial state,*
- $type : Q \to \{\exists, \forall\}$ *is the* state type mapping

and the transition function satisfies for every $q \in Q$ that all elements of $\delta(q, \rhd)$ are of form $(q', \rhd, R)$ for some $q' \in Q$.

All states q with $type(q) = \exists$ are called *existential*, all states with $type(q) = \forall$ are called *universal*. We now describe the semantics of the ATMs by first defining a configuration of an ATM, then a relation that describes possible computation steps between two configurations and finally a predicate that describes whether a configuration is accepting or not. In the following definitions, suppose that we have an arbitrary ATM $M = (Q, \Sigma, \Gamma, \sqcup, \rhd, \delta, q_0, type)$.

Definition 4.3 (Configuration). *A* configuration *of the alternating Turing machine M is a triple $(q, i, u) \in Q \times \mathbb{N} \times \Gamma^\omega$, where q is the* current state, *i is the* current position of the head, *and u is the* content of the tape.

For a set A, A^ω denotes the set of all infinite words over the alphabet A.

Definition 4.4 (Computation step relation). *A* computation step relation $\vdash$ *of the alternating Turing machine M is the binary relation between two configurations of M defined by*

$$(q, i, uxv) \vdash (q', i', ux'v) \iff \begin{cases} i' = i - 1 \text{ and } (q', x', L) \in \delta(q, x), \text{ or} \\ i' = i + 1 \text{ and } (q', x', R) \in \delta(q, x) \end{cases}$$

for all $q, q' \in Q$, $i, i' \in \mathbb{N}, u \in \Gamma^i, x \in \Gamma, v \in \Gamma^\omega$.

Definition 4.5 (Acceptance). *A configuration $c = (q, i, u)$ of the Turing machine M is called* accepting *if one of the following is true:*

- *the state q is an existential state and there exists at least one accepting configuration c' such that $c \vdash c'$,*
- *the state q is a universal state and all configurations c' such that $c \vdash c'$ are accepting.*

Special accepting and rejecting states arec not needed: each universal state with no successors is accepting; each existential state with no successors is rejecting.

The alternating Turing machine M accepts the input $w \in \Sigma^*$ if and only if the initial configuration $(q_0, 0, \rhd \cdot w \cdot \sqcup^\omega)$ is accepting.

Intuitively, existential states behave like states of a nondeterministic Turing machine: a run passing through an existential state continues to one of the possible successors. In contrast to this, a run entering a universal state forks and continues to all possible successors.

An ATM defines for each input a *computation tree*, in which the vertices are all the reachable configurations, the root vertex is the initial configuration, and there is an edge between each pair of configurations c and c' such that $c \vdash c'$. In order for this to really define a tree rather than a general graph, repeated occurrences of each configuration c have to be regarded as separate vertices in the configuration tree. If the configuration c is of form $c = \rhd \cdot u \cdot \sqcup^{\omega}$ for some $u \in \Gamma^k$, we say that the configuration c uses k units of space.

We now define several complexity classes related to alternating Turing machines. Computations in such complexity classes are bounded not only by time and space, but also by the number of alternations between existential and universal states during the computation. Although we need only complexity classes related to ATMs that are bounded in time and the number of alternations in this thesis, we define for completeness also the complexity classes that bound *space* and the number of alternations. Therefore, the following definition introduces two families of complexity classes.

Definition 4.6. *Let $f, g \colon \mathbb{N} \to \mathbb{N}$ be functions such that $g(n) \geq 1$. We define the complexity class* **ATIME**(f, g) *as the class of all problems A for which there is an alternating Turing machine that decides A and, for each input of length n, its computation tree has depth at most $f(n)$ and every branch contains at most $g(n) - 1$ alternations between existential and universal states.*

Similarly, the complexity class **ASPACE**(f, g) *is the class of all problems A for which there is an alternating Turing machine that decides A and, for each input of length n, all vertices in the computation tree use at most $f(n)$ units of space and its every branch contains at most $g(n) - 1$ alternations.*

If F and G are classes of functions, let

$$\mathbf{ATIME}(F, G) = \bigcup_{f \in F, g \in G} \mathbf{ATIME}(f, g) \textit{ and}$$
$$\mathbf{ASPACE}(F, G) = \bigcup_{f \in F, g \in G} \mathbf{ASPACE}(f, g).$$

If any of the parameters in the previous definitions should be unbounded, we write $*$ in its stead. For example, **ATIME**$(\mathcal{O}(n), *)$ is the class of all problems solvable by an ATM using linear time and arbitrarily many alternations. Note that **ATIME**$(F, *)$ is always equal to **ATIME**(F, F), as the ATM cannot make more alternations than computation steps.

Chandra et al. have observed several relationships between classical complexity classes related only to time or space and the complexity classes defined by ATMs [CKS81]. We recall some of these relationships:

Recall that $n^{\mathcal{O}(1)}$ is the class of all polynomial functions and $2^{n^{\mathcal{O}(1)}}$ is the class of all (single) exponential functions.

$$\mathbf{P} = \mathbf{ASPACE}(\mathcal{O}(\log n), *),$$
$$\mathbf{PSPACE} = \mathbf{ATIME}(n^{\mathcal{O}(1)}, *),$$
$$\mathbf{EXPTIME} = \mathbf{ASPACE}(n^{\mathcal{O}(1)}, *),$$
$$\mathbf{EXPSPACE} = \mathbf{ATIME}(2^{n^{\mathcal{O}(1)}}, *).$$

Relationships of some of the alternating complexity classes and the classes **NEXPTIME** and **EXPSPACE** are particularly important for this chapter. It can easily be seen that the class **NEXPTIME** corresponds to all problems solvable by an alternating Turing machine that starts in an existential state and can use exponential time and no alternations: this yields the inclusion

$$\mathbf{NEXPTIME} \subseteq \mathbf{ATIME}(2^{n^{\mathcal{O}(n)}}, 1).$$

On the other hand, results of Chandra et al. imply that the class **EXPSPACE** is precisely the complexity class **ATIME**$(2^{n^{\mathcal{O}(1)}}, 2^{n^{\mathcal{O}(1)}})$ of problems solvable in exponential time and with exponential number of alternations. An interesting class that lies in between those two complexity classes can be obtained by bounding the number of steps exponentially and the number of alternations polynomially. This class is called **AEXP**(poly).

Definition 4.7. **AEXP**(poly) = **ATIME**$(2^{n^{\mathcal{O}(1)}}, n^{\mathcal{O}(1)})$.

The following inclusions immediately follow from the mentioned results. However, it is unknown whether any of the inclusions is strict:

$$\textbf{NEXPTIME} \subseteq \textbf{AEXP}(\text{poly}) \subseteq \textbf{EXPSPACE}.$$

4.3 COMPLEXITY OF BV2 SATISFIABILITY

In this section, we show that the BV2 satisfiability problem is **AEXP**(poly)-complete. First, we prove that the problem is in the class **AEXP**(poly).

Theorem 4.1. *The* BV2 *satisfiability problem is in* **AEXP**(poly).

Proof. We describe the alternating Turing machine solving the problem. For a given BV2 formula φ, the machine first converts the formula to the prenex normal form, which can be done in polynomial while introducing only polynomially many new bit-vector variables (see Section 2.3.6). The machine then assigns values to all bits of all existentially quantified variables using existential states and to all bits of all universally quantified variables using universal states. Although this requires exponential time, as there are exponentially many bits whose value has to be assigned, only a polynomial number of alternations is required, because the formula φ can contain only polynomially many quantifiers.

Finally, the machine uses the assignment to evaluate the quantifier-free part of the formula. If the result of the evaluation is true, the machine accepts; it rejects otherwise. The evaluation takes exponential time and no quantifier alternations: the machine replaces all variables by exponentially many previously assigned bits and computes results of all operations from the bottom of the syntactic tree of the formula up. The computation of each of the operations takes time polynomial in the number of bits of its arguments, which is exponential in the size of the input formula. □

In the rest of this section, we show that the BV2 satisfiability problem is also hard for **AEXP**(poly). In particular, we present a reduction of a known **AEXP**(poly)-hard *second-order Boolean formula satisfiability problem* [Loh12; Lüc16] to the BV2 satisfiability.

Intuitively, the *second-order Boolean logic* (SO_2) can be obtained from a quantified Boolean logic by adding function symbols and quantification over such symbols. Alternatively, the SO_2 logic corresponds to the second-order predicate logic restricted to the domain $\{0, 1\}$. Lohrey and Lück have independently shown that by bounding the number of quantifier alternations in second-order Boolean formulas, problems complete for all levels of the exponential hierarchy can be obtained. Moreover, if the number of quantifier alternations is unbounded, the problem of deciding satisfiability of quantified second-order Boolean formulas is **AEXP**(poly)-complete [Loh12; Lüc16].

Exponential hierarchy is analogous to the polynomial hierarchy; the only difference is that the time bound on the alternating machine is exponential.

We now introduce the SO_2 logic more formally. The definitions of the syntax and semantics of SO_2 used in this chapter are due to Hannula et al. [Han+16].

Definition 4.8 (SO_2 syntax [Han+16]). *Let $\mathcal{F}$ be a countable set of function symbols, where each symbol $f \in \mathcal{F}$ is given an arity* $\operatorname{ar}(f) \in \mathbb{N}$. *The set* $\mathsf{SO}_2(\mathcal{F})$ *of* quantified Boolean second-order formulas *is defined inductively as*

$$\varphi ::= \varphi \wedge \varphi \mid \neg\varphi \mid \exists f \varphi \mid \forall f \varphi \mid f(\underbrace{\varphi, \dots, \varphi}_{\operatorname{ar}(f)\ \textit{times}}),$$

where $f \in \mathcal{F}$.

Definition 4.9 (SO_2 semantics [Han+16]). *An $\mathcal{F}$-*interpretation *is a function $\mathcal{I}$ that assigns to each symbol $f \in \mathcal{F}$ a Boolean function of the corresponding arity, i.e. $\mathcal{I}(f)\colon \{0,1\}^{\operatorname{ar}(f)} \to \{0,1\}$ for each $f \in \mathcal{F}$. We define the valuation of a formula $\varphi \in \mathsf{SO}_2(\mathcal{F})$ in $\mathcal{I}$, written $[\![\varphi]\!]_{\mathcal{I}}$, which is a number from $\{0,1\}$ computed recursively as*

$$\begin{aligned}
[\![\varphi \wedge \psi]\!]_{\mathcal{I}} &= [\![\varphi]\!]_{\mathcal{I}} \cdot [\![\psi]\!]_{\mathcal{I}},\\
[\![\neg\varphi]\!]_{\mathcal{I}} &= 1 - [\![\varphi]\!]_{\mathcal{I}},\\
[\![f(\varphi_1, \dots, \varphi_n)]\!]_{\mathcal{I}} &= \mathcal{I}(f)([\![\varphi_1]\!]_{\mathcal{I}}, \dots, [\![\varphi_n]\!]_{\mathcal{I}}),\\
[\![\exists f \varphi]\!]_{\mathcal{I}} &= \max\Big\{[\![\varphi]\!]_{\mathcal{I}[f \mapsto F]} \mid F\colon \{0,1\}^{\operatorname{ar}(f)} \to \{0,1\}\Big\},\\
[\![\forall f \varphi]\!]_{\mathcal{I}} &= \min\Big\{[\![\varphi]\!]_{\mathcal{I}[f \mapsto F]} \mid F\colon \{0,1\}^{\operatorname{ar}(f)} \to \{0,1\}\Big\}.
\end{aligned}$$

An SO_2 formula φ is satisfiable *if $[\![\varphi]\!]_{\mathcal{I}} = 1$ for some $\mathcal{I}$.*

We call function symbols of arity 0 *propositions* and all the other function symbols *proper functions*. An SO_2 formula φ is in the *prenex normal form* if it has the form $\overline{Q}\psi$, where $\overline{Q} = Q_1 f_1 Q_2 f_2 \dots Q_n f_n$ is a sequence of quantifiers and variables called a *quantifier prefix*, ψ is a quantifier-free formula called a *matrix*, and all proper functions are quantified before all propositions. In the following, we fix an arbitrary countable set of function symbols $\mathcal{F}$ and instead of $\mathsf{SO}_2(\mathcal{F})$, we write only SO_2.

Definition 4.10 (SO_2 satisfiability problem). *The* SO_2 satisfiability problem *is to decide whether a given closed SO_2 formula in the prenex normal form is satisfiable.*

Theorem 4.2 ([Loh12; Lüc16]). *The SO_2 satisfiability problem is* **AEXP**(poly)-*complete.*

We now show a polynomial time reduction of SO_2 satisfiability to BV2 satisfiability and thus finish the main claim of this chapter, which states that the BV2 satisfiability problem is **AEXP**(poly)-complete.

In this reduction, we use an *indexing* bit-vector operation, which is a special case of the extraction operation that produces only a single bit. In particular, for a bit-vector term $t^{[n]}$ and a number $0 \leq i < n$, the indexing operation $t^{[n]}[i]$ is defined as $\mathtt{extract}_{i,i}(t^{[n]})$. Recall that bits of bit-vectors are indexed from the least significant. For example, given a bit-vector variable $x^{[6]} = x_5 x_4 x_3 x_2 x_1 x_0$, the value of $x^{[6]}[1]$ refers to x_1. Although this operation was also used in reductions of Kovásznai et al. [KFB16], we need a more general version of the indexing operation, in which the index can be an arbitrary bit-vector term, not only a fixed scalar. This operation can be defined by using the indexing operation and the bit-shift operation with only a constant increase in the size of the term:

$$t^{[n]}[s^{[n]}] \;\equiv\; (t^{[n]} \gg s^{[n]})[0] \;\equiv\; \mathtt{extract}_{0,0}(t^{[n]} \gg s^{[n]}).$$

Moreover, the constant does not depend on the arguments of the indexing operation.

Theorem 4.3. *The* BV2 *satisfiability problem is* **AEXP**(poly)*-hard.*

Proof. We present a polynomial time reduction of the SO_2 satisfiability problem to the BV2 satisfiability problem. Let φ be an SO_2 formula with a quantifier prefix $\overline{Q}$ and a matrix ψ, i.e. $\varphi = \overline{Q}\psi$ where ψ is a quantifier-free formula. We construct a bit-vector formula φ^{BV} such that φ is satisfiable iff the formula φ^{BV} is satisfiable.

In the formula φ^{BV}, each function symbol f of the formula φ will be represented by a bit-vector variable x_f of bit-width $2^{\mathrm{ar}(f)}$. Intuitively, the individual bits of the variable x_f will encode values $f(b_{n-1}, \dots, b_0)$ for all possible inputs $b_0, \dots, b_{n-1} \in \{0, 1\}$. In particular, the value $f(b_{n-1}, \dots, b_0)$ is represented as the bit on the index $\sum_{i=0}^{n-1}(2^i b_i)$ in the bit-vector x_f. Equivalently, this index can be expressed as the numerical value of the bit-vector $b_{n-1} b_{n-2} \dots b_0$. For example, for a ternary function symbol f, bits of the bit-vector value $x_f = x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$ will represent values $f(1,1,1)$, $f(1,1,0)$, $f(1,0,1)$, $f(1,0,0)$, $f(0,1,1)$, $f(0,1,0)$, $f(0,0,1)$, and $f(0,0,0)$, respectively.

The reduction proceeds in two steps. First, we inductively construct a bit-vector term ψ^{BV} of bit-width 1, which corresponds to the formula ψ:

- If $\psi \equiv \rho_1 \wedge \rho_2$, we set $\psi^{BV} \equiv \rho_1^{BV} \mathbin{\&} \rho_2^{BV}$.
- If $\psi \equiv \neg\rho$, we set $\psi^{BV} \equiv {\sim}\rho^{BV}$.
- If $\psi \equiv f()$ (i.e., f is a proposition), we set $\psi^{BV} \equiv x_f^{[1]}$.
- If $\psi \equiv f(\rho_{n-1}, \dots, \rho_0)$ where $n = \mathrm{ar}(f)$, we set

  $$\psi^{BV} \equiv x_f^{[2^n]}\left[0^{[2^n - n]} \cdot \rho_{n-1}^{BV} \cdot \rho_{n-2}^{BV} \cdot \dots \cdot \rho_0^{BV}\right].$$

  Note that because both arguments of the indexing operation have to be of the same sort, $2^n - n$ zero bits have to be added to the index term to get a term of the same bit-width as the term $x_f^{[2^n]}$.

In the second step, we replace each quantifier $Q_i f_i$ in the quantifier prefix $\overline{Q}$ by a bit-vector quantifier $Q_i x_{f_i}^{[2^n]}$, where $n = \mathrm{ar}(f_i)$, and thus obtain a sequence of bit-vector quantifiers $\overline{Q}^{BV}$. The final formula φ^{BV} is then $\overline{Q}^{BV}$ $(\psi^{BV} = 1^{[1]})$.

Due to the binary representation of the bit-widths, the size of the formula φ^{BV} is polynomial in the size of the formula φ. Indeed, the input formula φ contains only polynomially many occurrences of function symbols and the size of the input formula is increased

- by $L(2^n) = n+1$ for each quantified variable of arity n due to converting $Q_i f_i$ to $Q_i x_{f_i}^{[2^n]}$,
- by $L(1) = 1$ for each occurrence of a proposition in the matrix due to converting $f()$ to $x_f^{[1]}$,
- by $L(2^n) + L(2^n - n) + L(0) + n = (\lfloor \log_2(2^n) \rfloor + 1) + (\lfloor \log_2(2^n - n) \rfloor + 1) + 1 + n \leq 3n + 3$ for each occurrence of a proper function symbol f of the arity n in the matrix due to converting f to $x_f^{[2^n]}$ and adding the numeral $0^{[2^n - n]}$ and n concatenation operations,
- by a constant for each occurrence of a proper function symbol due to using the generalized indexing operation, and
- by $1 + 1 + L(1) = 3$ due to modifying ψ^{BV} to $\psi^{BV} = 1^{[1]}$. □

Example 4.1. *Consider an* SO_2 *formula*

$$\exists f \forall p \forall q \,.\, \neg f(p, q, p) \wedge f(p, q \wedge \neg q, \neg p),$$

where f is a ternary function symbol and p, q are propositions. Then the result of the described reduction is the formula

$$\begin{aligned}&\exists x_f^{[8]} \forall x_p^{[1]} \forall x_q^{[1]} \Big(\\ &\quad (\sim x_f^{[8]}[0^{[5]} \cdot x_p^{[1]} \cdot x_q^{[1]} \cdot x_p^{[1]}] \mathbin{\&} x_f^{[8]}[0^{[5]} \cdot x_p^{[1]} \cdot (x_q^{[1]} \mathbin{\&} \sim x_q^{[1]}) \cdot \sim x_p^{[1]}])\\ &\qquad = 1^{[1]}\Big).\end{aligned}$$

Note that the resulting formula is satisfiable and the witness to its satisfiability is the assignment $\mu(x_f) = 00010010$. *In the original formula, this assignment corresponds to the following function f:*

$$\begin{aligned}
f(0,0,0) &= \mu(x_f)_0 = 0, & f(0,0,1) &= \mu(x_f)_1 = 1,\\
f(0,1,0) &= \mu(x_f)_2 = 0, & f(0,1,1) &= \mu(x_f)_3 = 0,\\
f(1,0,0) &= \mu(x_f)_4 = 1, & f(1,0,1) &= \mu(x_f)_5 = 0,\\
f(1,1,0) &= \mu(x_f)_6 = 0, & f(1,1,1) &= \mu(x_f)_7 = 0.
\end{aligned}$$

Observe that the function thus defined is indeed a witness of satisfiability of the original formula.

Corollary 4.1. *The* $\mathsf{BV2}$ *satisfiability problem is* **AEXP**(poly)*-complete.*

	Quantifiers			
	No		Yes	
	Uninterpreted functions		Uninterpreted functions	
Encoding	No	Yes	No	Yes
Unary	**NP**	**NP**	**PSPACE**	**NEXPTIME**
Binary	**NEXPTIME**	**NEXPTIME**	**AEXP**(poly)	**2–NEXPTIME**

Table 4.2: Completeness results for various bit-vector logics and encodings. This is the table presented by Fröhlich et al. [FKB13] extended by the result proved in this chapter.

4.4 CONCLUSIONS

We have identified the precise complexity class of deciding satisfiability of quantified bit-vector formulas with binary-encoded bit-widths. This chapter shows that the problem is complete for the complexity class **AEXP**(poly); i.e., the class of all problems solvable by an alternating Turing machine that can use exponential time and a polynomial number of alternations. This result had settled the open question raised by Kovásznai et al. [KFB16]. Known completeness results for various bit-vector logics including the result proven in this chapter are summarized in Table 4.2.

5 SOLVING QUANTIFIED BIT-VECTOR FORMULAS BY BINARY DECISION DIAGRAMS

This chapter presents an algorithm for solving satisfiability of quantified bit-vector formulas that is based on binary decision diagrams. Although binary decision diagrams have been previously used to implement satisfiability decision procedures for the propositional logic, state-of-the-art solvers based on the Conflict Driven Clause Learning algorithm (CDCL) [MSS99] usually achieve much better performance. The main disadvantage of BDDs is low scalability: the size of a BDD corresponding to a propositional formula can be exponential in the number of propositional variables, and when a BDD becomes too large, some operations are very slow. Employment of BDDs in SMT solving makes more sense when formulas with quantifiers are considered: quantification usually reduces size of a BDD as it decreases the number of BDD variables. This can be documented by Figure 5.1, which compares the BDD sizes for formulas before and after existential or universal quantification on a subset of formulas from the SMT-LIB benchmark repository.

As a related work for using BDDs for formulas with quantifiers, there already exist some BDD-based tools deciding validity of quantified Boolean formulas with the performance similar to state-of-the-art solvers for this problem [OE11; AS04].

The main idea of using BDDs for solving satisfiability of quantified bit-vector formulas is simple: the formula is satisfiable if and only if the corresponding BDD is not $\boxed{0}$. Even this approach without any improvements can solve some formulas that cannot be solved by the state-of-the-art SMT solver Z3, which uses model-based quantified-instantiation approach, explained in Section 3.1. In particular, consider the unsatisfiable formula

$$\begin{aligned} \varphi \;\equiv\; & \left(x^{[32]} = (16^{[32]} \times y^{[32]}) + (16^{[32]} \times z^{[32]})\right) \wedge \\ & \forall v^{[32]} \left(x^{[32]} \neq 16^{[32]} \times v^{[32]}\right), \end{aligned}$$

which was discussed in Section 3.1. The BDD for the subformula $\forall v^{[32]} (x^{[32]} \neq 16^{[32]} \times v^{[32]})$ can be easily computed by the function `f2BDD` and is shown in Figure 5.2. The BDD shows that the subformula is satisfiable if and only if one of the four least significant bits of x is 1. After computing the BDD for the entire formula φ, one gets the BDD $\boxed{0}$, and hence the formula φ can be decided as unsatisfiable.

Our BDD-based algorithm for satisfiability of the bit-vector formulas consists of three main components, thanks to which the described approach scales well for more complex formulas:

- Formula simplifications, which reduce the number of variables in the formula and push quantifiers downwards in the syntax tree of the formula (which later helps to keep intermediate BDDs smaller as they are

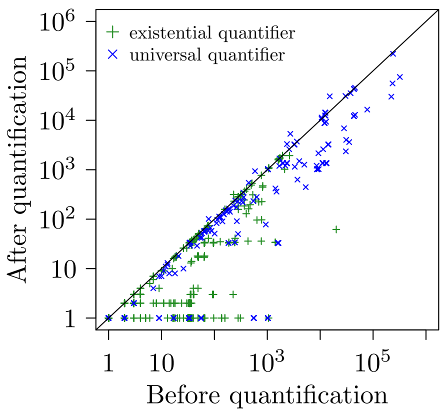


Figure 5.1: Comparison of sizes (measured by the number of BDD nodes) of BDDs corresponding to all quantified subformulas in quantified bit-vector benchmarks from the family *wintersteiger* in the SMT-LIB benchmark repository, before and after quantification.

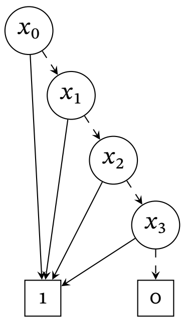


Figure 5.2: A BDD for $\forall v^{[32]}\,(\neg(x^{[32]} = 2^{4^{[32]}} \times v^{[32]}))$.

built in the bottom-up manner). Formula simplifications can reduce some formulas to $\top$ or $\bot$ and thus immediately decide their satisfiability.

- Construction of a BDD using a specific variable ordering. The ordering has a significant influence on the BDD size.

- Formula approximations, which represent some bit-vector variables in the formula by a smaller number of bits and thus lead to smaller BDDs. Unsatisfiability of an overapproximation of a formula implies unsatisfiability of the original formula and, dually, satisfiability of a formula underapproximation implies satisfiability of the original formula.

We present a minor contribution in each component. The main contribution of our approach is the fact that the algorithm based on the three parts can compete with leading SMT solvers for the bit-vector theory, namely Boolector, CVC4, and Z3. This claim is supported by the experimental evaluation in Chapter 9.

This chapter introduces the main components step by step. Section 5.1 describes the simplifications, Section 5.2 describes how to compute the ordering of BDD variables, and Section 5.3 describes formula approximations. Finally, Section 5.4 presents our complete algorithm.

5.1 FORMULA SIMPLIFICATIONS

As in most of the modern SMT solvers, the first step of deciding satisfiability is simplification of the input formula. Besides trivial simplifications (e.g., $\varphi \wedge \varphi$ reduces to φ), we apply the following simplification rules.

MINISCOPING. Miniscoping [Har09] is a technique for reducing scopes of universal quantifiers by: (i) distributing universal quantifiers over conjunctions and (ii) quantifying only one of the disjuncts, if the other disjunct does has no free occurences of the quantified variable. Existential quantifiers are handled analogously. The simplification rules are as follows:

$$\begin{aligned} \forall x\,(\varphi \wedge \psi) &\rightsquigarrow (\forall x\,\varphi) \wedge (\forall x\,\psi),\\ \exists x\,(\varphi \vee \psi) &\rightsquigarrow (\exists x\,\varphi) \vee (\exists x\,\psi), \end{aligned}$$

and for formulas ψ with no free occurrences of the variable x:

$$\begin{aligned} \forall x\,(\varphi \vee \psi) &\rightsquigarrow (\forall x\,\varphi) \vee \psi,\\ \exists x\,(\varphi \wedge \psi) &\rightsquigarrow (\exists x\,\varphi) \wedge \psi. \end{aligned}$$

The miniscoping technique is beneficial for BDD-based SMT solvers because the earlier application of the quantification potentially allows earlier reduction of the BDD size.

DESTRUCTIVE EQUALITY RESOLUTION. Destructive equality resolution (DER) [WHM13] eliminates a universally quantified variable x in a formula of form $\forall x\,(x \neq t \vee \varphi)$, where t is a term that does not contain the variable x. The formula is equivalent to $\forall x\,(x = t \rightarrow \varphi)$ and hence also to $\varphi[x \leftarrow t]$. The simplification rule is formulated as follows:

$$\forall x\,(x \neq t \vee \varphi) \rightsquigarrow \varphi[x \leftarrow t].$$

CONSTRUCTIVE EQUALITY RESOLUTION. Constructive equality resolution (CER) is a dual version of DER. As far as we know, it was not considered before as solvers for quantified formulas typically work with formulas after Skolemization and thus without any existential quantifiers. Constructive equality resolution can be formulated as the following simplification rule, where t has the same meaning as above:

After we introduced CER, it was adopted also by Boolector [Pre17].

$$\exists x\,(x = t \wedge \varphi) \rightsquigarrow \varphi[x \leftarrow t].$$

Both DER and CER also tend to produce smaller BDDs because they eliminate one bit-vector variable and, in turn, also eliminate several BDD variables.

PURE LITERAL ELIMINATION Pure literal elimination eliminates Boolean variables that occur in the formula only with one polarity. Namely, if an existentially quantified Boolean variable occurs only positively, it can be replaced by $\top$; if it occurs only negatively, it can be replaced by $\bot$. The treatment is dual for universally quantified variables. This gives the following simplification rules:

$$\begin{aligned} \exists x^{Bool}\,\varphi &\rightsquigarrow \varphi[x^{Bool} \leftarrow \top], \\ \forall x^{Bool}\,\varphi &\rightsquigarrow \varphi[x^{Bool} \leftarrow \bot], \end{aligned}$$

if all the occurrences of x^{Bool} in φ have the positive polarity and

$$\begin{aligned} \exists x^{Bool}\,\varphi &\rightsquigarrow \varphi[x^{Bool} \leftarrow \bot], \\ \forall x^{Bool}\,\varphi &\rightsquigarrow \varphi[x^{Bool} \leftarrow \top], \end{aligned}$$

if all the occurrences of x^{Bool} in φ have the negative polarity.

THEORY-RELATED SIMPLIFICATIONS. We also perform several simplifications related to the interpretation of the function and predicate symbols in the bit-vector theory. Examples of such simplifications are reductions

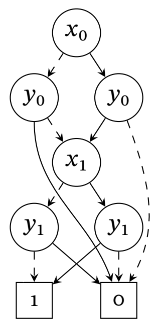
$$\begin{aligned} t^{[n]} + (-t^{[n]}) &\rightsquigarrow 0^{[n]}, \\ t^{[n]} \times 0^{[n]} &\rightsquigarrow 0^{[n]}, \\ t^{[t]} \,\&\, 0^{[n]} &\rightsquigarrow 0^{[n]}\text{ , or} \\ \mathtt{extract}_{j,i}(0^{[n]}) &\rightsquigarrow 0^{[j-i+1]}. \end{aligned}$$

Note that all the mentioned simplification rules have no effect on models of the formula and thus they have no direct effect on the resulting BDD. However, simplified formulas tend to have simpler subformulas and thus the intermediate BDDs are often smaller and the resulting BDD can be computed faster.

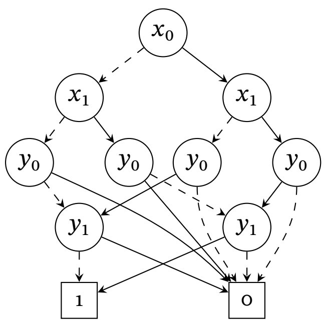
5.2 BIT VARIABLE ORDERING

When constructing a BDD, one has to specify an ordering of BDD variables, which in our case precisely correspond to bit variables of the input formula. Although there are algorithms for changing the ordering of the variables during the computation, the reordering can be costly and therefore choosing the right variable upfront is crucial. The ordering of variables has a significant effect on the BDD size and the time that is necessary to compute it. In some cases, the size of a BDD for a formula is linear with respect to the number of bits in the formula with one variable ordering, but exponential with another one.

For example, consider the formula $\varphi_1 \equiv (x^{[n]} = y^{[n]})$ for an arbitrary $n \in \mathbb{N}^+$ and let $x_0, x_1, \dots, x_{n-1}$ be the bits of x and $y_0, y_1, \dots, y_{n-1}$ be the bits of y. We define two orderings:



(a) Ordering $\leq_1$



(b) Ordering $\leq_2$

Figure 5.3: Examples of BDDs representing the bit-vector formula $x^{[2]} = y^{[2]}$ with different variable orderings.

$\leq_1$ All bit variables are ordered according to their significance (from the least to the most significant) and variables with the same significance are ordered by the order of the first occurrence of the corresponding bit-vector variables in the formula. For the considered formula φ_1, we get:

$$x_0 \leq_1 y_0 \leq_1 x_1 \leq_1 y_1 \leq_1 \dots \leq_1 x_{n-1} \leq_1 y_{n-1}.$$

$\leq_2$ Bit variables are ordered by the order of the first occurrence of the corresponding bit-vector variable in the formula and bit variables corresponding to the same bit-vector variable are ordered according to their significance (from the least to the most significant). For the considered formula, we get:

$$x_0 \leq_2 x_1 \leq_2 \dots \leq_2 x_{n-1} \leq_2 y_0 \leq_2 y_1 \leq_2 \dots \leq_2 y_{n-1}.$$

It can be seen that the BDD for φ_1 using the ordering $\leq_1$ has $3n+2$ nodes, while the BDD for the same formula and $\leq_2$ has $3 \cdot 2^n - 1$ nodes. Figures 5.3a and 5.3b show these BDDs for $n = 2$ and orderings $\leq_1$ and $\leq_2$, respectively.

However, these orderings can lead to opposite results with other formulas. For example, the size of the BDD for the formula

$$\begin{aligned}\varphi_2 \equiv\ & \left(x_1{}^{[2]} = (x_2{}^{[2]} \gg_u 1^{[2]})\right) \wedge \\ & \left(x_3{}^{[2]} = (x_4{}^{[2]} \gg_u 1^{[2]})\right) \wedge \\ & \dots \wedge \\ & \left(x_{2n-1}{}^{[2]} = (x_{2n}{}^{[2]} \gg_u 1^{[2]})\right)\end{aligned}$$

using the ordering $\leq_1$ is $2^{n+2} - 1$, while it is only $4n + 2$ for $\leq_2$. In the following, we introduce an ordering $\leq_3$, which in most cases combines advantages of both $\leq_1$ and $\leq_2$. However, we cannot hope that the introduced ordering will be optimal for all the cases as it is well known that choosing the optimal BDD variable ordering is an NP-complete problem [BW96].

Let *vars*(φ) be the set of bit-vector variables that appear in the input formula φ. We call the variables $x, y \in$ *vars*(φ) *potentially dependent*, written

$x \sim y$, if they both appear in the same atomic subformula of φ. Let $\simeq$ be the equivalence on $vars(\varphi)$ defined as the transitive closure of $\sim$. Every $v \in vars(\varphi)$ then defines an equivalence class $[v]_\simeq$ of transitively dependent variables. For example, in the formula φ_2, the relation $\simeq$ partitions the set $vars(\varphi)$ to the equivalence classes

$$\{\{{x_1}^{[2]}, {x_2}^{[2]}\}, \{{x_3}^{[2]}, {x_4}^{[2]}\}, \dots, \{{x_{2n-1}}^{[2]}, {x_{2n}}^{[2]}\}\}.$$

Using the relation $\simeq$, we define the following ordering $\leq_3$. Its idea is to have all potentially dependent variables near to each other, which can reduce the size of the BDD.

$\leq_3$ Equivalence classes of $\simeq$ are first ordered by the first occurrences of the bit-vector variables in φ. In particular for $u \not\simeq v$, $u_i \leq_3 v_j$ if there is a bit-vector variable in $[u]_\simeq$ that occurs in φ before all bit-vector variables of $[v]_\simeq$. Within the individual equivalence classes, bit variables are ordered according to $\leq_1$.

Note that for both formulas φ_1 and φ_2 mentioned above, $\leq_3$ coincides with the better of the orderings $\leq_1$ and $\leq_2$.

In addition to the initial variable ordering, there are several techniques that dynamically reorder the BDD variables to reduce the BDD size. We use *sifting* [Rud93] as usually the most successful one [Knu09].

5.3 APPROXIMATIONS

For some bit-vector formulas, e.g., formulas containing non-linear multiplication, the size of the BDD representation is exponential with respect to the number of bits for every possible variable ordering [Bry91]. This is the case even for simple formulas such as $x^{[32]} = y^{[32]} \times z^{[32]}$. Fortunately, satisfiability of these formulas can be often decided using their overapproximations or underapproximations. Given a formula φ, its *underapproximation* is any formula $\underline{\varphi}$ that logically entails φ, and its *overapproximation* is any formula $\overline{\varphi}$ logically entailed by φ. I.e.,

$$\underline{\varphi} \models \varphi \models \overline{\varphi}.$$

Clearly, every model of $\underline{\varphi}$ is also a model of φ and if an underapproximation $\underline{\varphi}$ is satisfiable, so is the formula φ. Similarly, if an overapproximation $\overline{\varphi}$ is unsatisfiable, so is φ.

Example 5.1. *The model-based quantifier instantiation presented in Section 3.1 can be also seen as a technique based on an iterative overapproximation refinement: the formulas φ, $\varphi \wedge \psi[x \leftarrow t_1]$, $\varphi \wedge \psi[x \leftarrow t_1] \wedge \psi[x \leftarrow t_2]$, etc. are overapproximations of the formula $\varphi \wedge \forall x\,(\psi)$.*

Different approximations can be found in SMT solvers for *quantifier-free* bit-vector formulas. For example, the SMT solver UCLID overapproximates a formula in the negation normal form by replacing some subformulas with fresh variables [Bry+07]. Both the underapproximations and overapproximations that are used in our algorithm are inspired by yet another approximation technique, which is used in SMT solvers UCLID and Boolector for quantifier-free bit-vector formulas. These solvers underapproximate a formula by restricting values of some bits of the chosen bit-vector variables while leaving the remaining bits unchanged [Bry+07; BB09]. The number of bit variables used to represent the bit-vector variable is called its *effective bit-width*. This technique can be seen in Example 5.2; in this example, 30 most significant bits of the variables $x^{[32]}$, $y^{[32]}$, and $z^{[32]}$ are restricted to the value 0 and only two effective bits are used to represent their two least significant bits.

Example 5.2. *Consider the formula* $\varphi \equiv x^{[32]} = y^{[32]} \times z^{[32]}$. *One of its underapproximations is the formula*

$$\begin{aligned}\underline{\varphi} \equiv\ & (x^{[32]}[31:2] = 0^{[30]}) \land \\ & (y^{[32]}[31:2] = 0^{[30]}) \land \\ & (z^{[32]}[31:2] = 0^{[30]}) \land \\ & (x^{[32]} = y^{[32]} \times z^{[32]}).\end{aligned}$$

Although the BDD for this formula is not exponential, it is infeasible to be computed directly since the BDD for the subformula $x^{[32]} = y^{[32]} \times z^{[32]}$ *is exponential. However, the formula* $\underline{\varphi}$ *is equivalent to the following formula* $\underline{\varphi}'$ *for which the computation is feasible:*

$$\begin{aligned}\underline{\varphi}' \equiv\ & (x^{[32]}[31:2] = 0^{[30]}) \land \\ & (y^{[32]}[31:2] = 0^{[30]}) \land \\ & (z^{[32]}[31:2] = 0^{[30]}) \land \\ & ((3^{[32]} \mathbin{\&} x^{[32]}) = (3^{[32]} \mathbin{\&} y^{[32]}) \times (3^{[32]} \mathbin{\&} z^{[32]})).\end{aligned}$$

To see the equivalence of $\underline{\varphi}$ *and* $\underline{\varphi}'$, *observe that the condition* $(x^{[32]}[31:2] = 0^{[30]})$ *implies that 30 most significant bits of* $x^{[32]}$ *are* 0 *and that the term* $(3^{[32]} \mathbin{\&} x^{[32]})$ *yields the value of* $x^{[32]}$ *after setting its 30 most significant bits to* 0.

Because the BDD for the formula $\underline{\varphi}'$ *is not* $\boxed{0}$, *the original formula* φ *is concluded to be satisfiable.*

However, there are multiple choices both for the bits that are represented by the effective bit-width and for the value to which all other bits are restricted. We now explore several of these possibilities, which are called *extensions* in the original UCLID and Boolector publications.

5.3.1 *Extensions*

Let $x^{[n]}$ be a bit-vector variable of bit-width n and $e \in \mathbb{N}^+$ be its desired effective bit-width. If $e \geq n$, the variable $x^{[n]}$ can be left unchanged. Otherwise, the variable $x^{[n]}$ has to be represented by a smaller number of effective bits. All extensions that we discuss are illustrated in Figure 5.4. The first two possible extensions come from UCLID and Boolector:

ZERO-EXTENSION uses the effective bit-width to represent the e least significant bits and sets the $n - e$ most significant bits to 0.

SIGN-EXTENSION also uses the effective bit-width to represent the e least significant bits, but sets the $n - e$ most significant bits to the value of the e-th least significant bit.

In English sentences, bits are indexed from 1, i.e., the least significant bit is the 1st least significant.

As opposed to the zero-extension, the sign-extension allows also representing negative numbers with small absolute value expressed in two's complement notation such as $sbv_4(-1) = 1111$ or $sbv_4(-2) = 1110$. Another extensions are suggested in [BB09], e.g., the *one-extension* defined analogously to the zero-extension. However, we do not consider the one-extension here because it produces only few zero bits, which are desired as they tend to reduce the size of BDDs for multiplication.

Although the zero-extension and the sign-extension already cover interesting corner cases, they require large effective bit-widths to cover corner case values as 10000000 or 01000000, which are often interesting due to overflows. To cover such values, we introduce the following *right* variants of the above-mentioned extensions, which use the effective bit-width to represent the most significant bits of the variable:

RIGHT ZERO-EXTENSION uses the effective bit-width to represent the e most significant bits and sets the $n - e$ least significant bits to 0.

RIGHT SIGN-EXTENSION also uses the effective bit-width to represent the e most significant bits, but sets the $n - e$ least significant bits to the value of the e-th most significant bit.

We also propose a combination of left- and right- extensions, which we call *middle extensions*. These use one half of the effective bit-width to represent the most significant bits and the other half of the effective bit-width to represent the least significant bits. Middle extensions can represent both the small values of the given variable and the mentioned corner cases such as 10000000 or 01000000. Moreover, they can represent values such as 01000010. On the other hand, middle extensions need the effective bit-width equal to the original bit-width to represent values such as 00010000.

In our original algorithm, this was achieved by iterating between left- and right- extensions instead of using the simpler method of middle extensions.

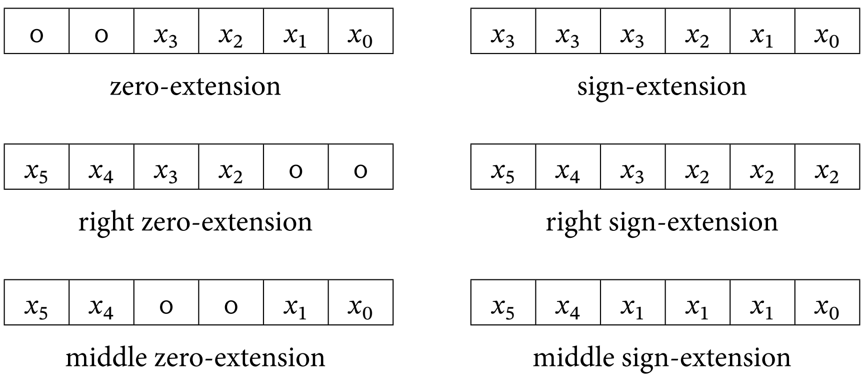

0 0 x_3 x_2 x_1 x_0
zero-extension
x_3 x_3 x_3 x_2 x_1 x_0
sign-extension
x_5 x_4 x_3 x_2 0 0
right zero-extension
x_5 x_4 x_3 x_2 x_2 x_2
right sign-extension
x_5 x_4 0 0 x_1 x_0
middle zero-extension
x_5 x_4 x_1 x_1 x_1 x_0
middle sign-extension

Figure 5.4: Reductions of $x^{[6]} = x_5x_4x_3x_2x_1x_0$ to 4 effective bits.

MIDDLE ZERO-EXTENSION uses the effective bit-width to represent the $\lfloor e/2 \rfloor$ most significant bits and $\lceil e/2 \rceil$ least significant bits, and sets the $n - e$ middle bits to 0.

MIDDLE SIGN-EXTENSION also uses the effective bit-width to represent the $\lfloor e/2 \rfloor$ most significant bits and $\lceil e/2 \rceil$ least significant bits, but sets the $n-e$ middle bits to the value of the $(\lceil e/2 \rceil)$-th least significant bit.

Similarly to Example 5.2, reduction of the effective bit-width of a variable in a *quantifier-free* formula using any of the described extensions is simple to implement. The input formula can be conjoined with a suitable formula that restricts the values of the chosen bits. Let $x^{[n]}$ be a variable whose bit-width should be reduced and $1 \leq e < n$ the desired effective bit. We describe for each extension a formula $\rho^{e}_{x^{[n]}}$ such that $\underline{\varphi} = \rho^{e}_{x^{[n]}} \wedge \varphi$ is the desired underapproximation of the input formula φ.

- For the (left) zero-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[n-1:e] = 0^{[n-e]}.$$
- For the (left) sign-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[n-1:e] = \texttt{signExtend}_{n-e-1}(x^{[n]}[e-1]).$$
- For the right zero-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[n-e-1:0] = 0^{[n-e]}.$$
- For the right sign-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[n-e-1:0] = \texttt{signExtend}_{n-e-1}(x^{[n]}[n-e]).$$
- For the middle zero-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[(n-\lfloor e/2 \rfloor - 1):\lceil e/2 \rceil] = 0^{[n-e]}.$$
- For the middle sign-extension, $\rho^{e}_{x^{[n]}}$ is
$$x^{[n]}[(n-\lfloor e/2 \rfloor - 1):\lceil e/2 \rceil] = \texttt{signExtend}_{n-e-1}(x^{[n]}[\lceil e/2 \rceil - 1]).$$

Recall that $t[i]$ was defined as $t[i:i]$.

As was discussed in Example 5.2, if this approach should be beneficial for BDD-based SMT solvers, each original occurrence of a variable $x^{[n]}$ in the formula $\underline{\varphi}$ must also be modified to reflect the restricted values of its bits. For example, if the zero-extension was performed, $n - e$ most significant bits of the variable $x^{[n]}$ have to be fixed to 0; if the sign-extension was performed, $n - e$ most significant bits of the variable $x^{[n]}$ have to be fixed to the most significant effective bit of $x^{[n]}$. Such modifications simplify the computations that depend on the restricted value of $x^{[n]}$. We will, however, not consider these modifications during proofs of the correctness of the approach, because the result of these modifications is the formula that is equivalent to the formula $\underline{\varphi}$, albeit it is potentially simpler to compute.

5.3.2 *Approximations of quantified formulas*

The reduction of effective bit-width can be performed also in quantified formulas. Moreover, in quantified formulas, both underapproximations and overapproximations of the input formula can be obtained by reducing effective bit-widths of a suitable subset of variables. In the following, we fix one of the six introduced extension methods and refer to it only as *the chosen extension*. We also suppose that the input formula is in the negation normal form and does not contain free variables.

To obtain an underapproximation of a formula, we reduce the effective bit-width of all its *existentially quantified* variables. Given a formula φ and $e \in \mathbb{N}^+$, let $\underline{\varphi}_e$ denote the formula φ with the effective bit-width of each existentially quantified variable reduced to e by the chosen extension. Namely, the formula $\underline{\varphi}_e$ is obtained from the formula φ by replacing each its subformula of the form $\exists x^{[n]}\,(\psi)$ by the formula $\exists x^{[n]}\,(\rho^e_{x^{[n]}} \wedge \psi)$.

On the other hand, to obtain an overapproximation of a formula, we reduce the effective bit-widths of all its *universally quantified* variables. Given a formula φ and $e \in \mathbb{N}^+$, let $\overline{\varphi}_e$ denote the formula φ with the effective bit-width of each universally quantified variable reduced to e by the chosen extension. Namely, the formula $\overline{\varphi}_e$ is obtained from the formula φ by replacing each its subformula of the form $\forall x^{[n]}\,(\psi)$ by the formula $\forall x^{[n]}\,(\rho^e_{x^{[n]}} \rightarrow \psi)$.

The following theorem establishes that for each formula φ in the NNF, $\underline{\varphi}_e$ is indeed its underapproximation and $\overline{\varphi}_e$ is its overapproximation.

Theorem 5.1. *For every formula φ in the NNF and any $e \in \mathbb{N}^+$, it holds:*

1. *If μ is a model of $\underline{\varphi}_e$, then μ is also a model of φ.*
2. *If μ is a model of φ, then μ is also a model of $\overline{\varphi}_e$.*

Proof. We show the first claim by induction on the structure of the formula φ:

- If the formula φ is quantifier-free, the claim holds trivially since formulas φ and $\underline{\varphi}_e$ are identical.
- If the formula φ is of form $\exists x^{[n]}\,(\psi)$: let μ be a model of $\underline{\varphi}_e$, which is defined as $\underline{\varphi}_e = \exists x^{[n]}\,(\rho^e_{x^{[n]}} \wedge \underline{\psi}_e)$. Therefore there is a bit-vector v of bit-width n, such that $\mu[x^{[n]} \mapsto v]$ is a model of $\rho^e_{x^{[n]}} \wedge \underline{\psi}_e$. This assignment is thus also a model of $\underline{\psi}_e$ and from the induction hypothesis also of ψ. The assignment μ is hence a model of the formula $\exists x^{[n]}\,(\psi)$.
- If the formula φ is of form $\forall x^{[n]}\,(\psi_1)$, or $\psi_1 \wedge \psi_2$, or $\psi_1 \vee \psi_2$: In these cases, the arguments ψ_i are replaced by formulas $\underline{\psi_i}_e$. From the induction hypothesis we know that each formula $\underline{\psi_i}_e$ logically entails the formula ψ_i. The claim thus follows from the monotonicity of the operations $\forall$, $\wedge$, and $\vee$. In other words, if each argument ψ_i is replaced by a formula that logically entails ψ_i, the result also entails the original formula φ.

We do not consider the negations during the induction, because since the formula is in the NNF, negations can occur only in its quantifier-free part.

The proof of the second claim is analogous. □

An obvious corollary hence allows deciding satisfiability of φ by deciding satisfiability of $\underline{\varphi}_e$ or $\overline{\varphi}_e$.

Corollary 5.1. *For every formula φ in the NNF and any $e \in \mathbb{N}^+$, it holds:*

1. *If the formula $\underline{\varphi}_e$ is satisfiable, so is the formula φ.*
2. *If the formula $\overline{\varphi}_e$ is unsatisfiable, so is the formula φ.*

5.4 THE ALGORITHM

We now present the complete BDD-based algorithm for deciding satisfiability of quantified bit-vector formulas. In the algorithm, we use the procedure `f2BDD` from Preliminaries, which converts a formula to the corresponding BDD recursively on the structure of the formula.

For a given input formula φ, the algorithm proceeds in the following steps:

1. Simplify the formula φ using the rules discussed in Section 5.1 up to the fixed point and convert it to the negation normal form. If the result is $\top$, return `SAT`. If the result is $\bot$, return `UNSAT`.
2. Take the simplified formula in NNF φ' and compute a chosen ordering $\leq_i$ as described in Section 5.2. This ordering will be used as the initial ordering in the procedure `f2BDD`.
3. Run the following three threads in parallel. Return the first result `SAT` or `UNSAT` that any of the threads returns:
   a) *Precise solver*: Call `f2BDD(`φ'`)` to compute the BDD corresponding to φ'. If the returned BDD is $\boxed{0}$, return `UNSAT`. Otherwise return `SAT`.
   b) *Under-approximating solver*: Sequentially compute `f2BDD(`$\underline{\varphi'}_i$`)` for $i = 1, 2, 4, 6, 8, \ldots$ until reaching the greatest bit-width of any bit-vector variable in φ'. If any of the resulting BDDs is distinct from $\boxed{0}$, return `SAT`. Otherwise return `UNSAT`.
   c) *Over-approximating solver*: Sequentially compute `f2BDD(`$\overline{\varphi'}_i$`)` for $i = 1, 2, 4, 6, 8, \ldots$ until reaching the greatest bit-width of any bit-vector variable in φ'. If any of the produced BDDs is $\boxed{0}$, return `UNSAT`. Otherwise return `SAT`.

The high-level workflow of the algorithm can be found in Figure 5.5.

Note that the algorithm is parametrized by the choice of an ordering and the extension used for approximations. Regardless these parameters, the algorithm is sound and complete. However, in practice, the procedure `f2BDD` may need exponential time and memory and thus the algorithm may not finish within reasonable limits.

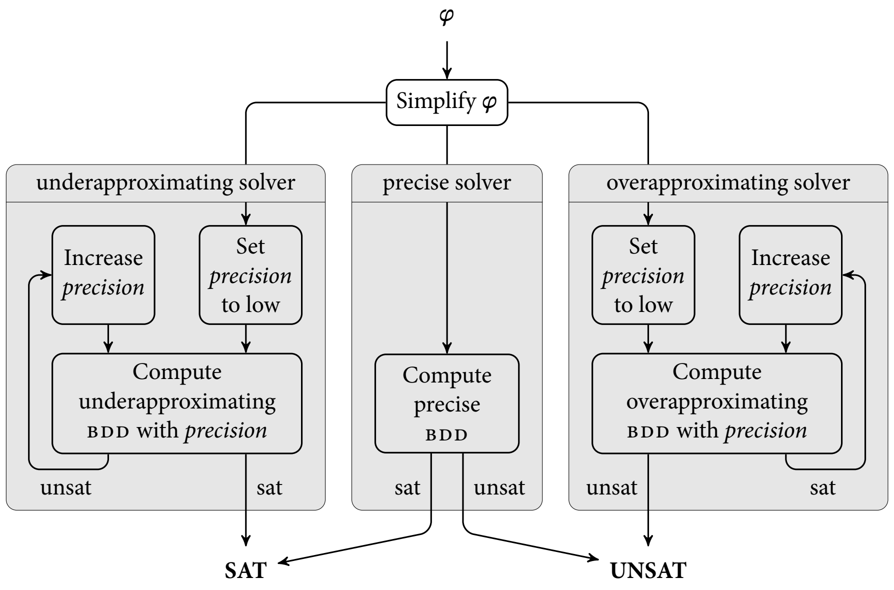


Figure 5.5: High-level overview of the algorithm for solving quantified bit-vector formulas by BDDs. The three shaded areas are executed in parallel and the first result is returned.

6 ABSTRACTION OF BIT-VECTOR OPERATIONS FOR BDD-BASED SMT SOLVERS

Although the algorithm presented in the previous chapter is competitive with state-of-the-art SMT solvers on most of the real-world formulas, the approach has several drawbacks. For example, the presented algorithm cannot solve satisfiability of simple formulas such as

$$\exists x, y \ ((x \times y = 0) \ \land \ (x <_u 2) \ \land \ (x >_u 4)),$$

$$\exists x, y \ ((x \ll 1) \times y = 1),$$

$$\exists x, y \ (x >_u 0 \ \land \ x \leq_u 4 \ \land \ y >_u 0 \ \land \ y \leq_u 4 \ \land \ x \times y = 0),$$

where all variables and constants have bit-width 32. All three of these formulas are unsatisfiable, but cannot be decided without approximations, because they contain non-linear multiplication. Moreover, the introduced approximations do not help as the formulas are unsatisfiable and contain no universally quantified variables, which could be used to approximate the formula.

However, the three above-mentioned formulas have something in common: only a few of the bits of the multiplication results are sufficient to decide satisfiability of the formulas. The first formula can be decided unsatisfiable without computing any bits of $x \times y$ whatsoever. The second formula can be decided by computing only the least significant bit of $(x \ll 1) \times y$ because it must always be zero. The third formula can be decided by computing 5 least significant bits of $x \times y$, because they are enough to rule out all values of x and y between 1 and 4 as models.

With this in mind, we propose an improvement of BDD-based SMT approaches, such as the one presented in the previous chapter, by allowing to compute only several bits of results of the selected arithmetic operations. To achieve this, the chapter defines abstract domains in which the operations can produce *do-not-know* values and shows that these abstract domains can be used to decide satisfiability of an input formula.

This chapter is structured as follows. Section 6.1 presents a general definition of abstract domains for terms and formulas and shows how they can be used to decide satisfiability of a formula. Section 6.2 introduces truncating term and formula abstract domains that compute only several bits from results of arithmetic bit-vector operations. Section 6.3 shows how to combine the proposed abstractions with the bit-width approximations, which were described in the previous chapter. And finally, Section 6.4 describes several further improvements to the application of these abstract domains.

6.1 FORMULA AND TERM ABSTRACTIONS

As was discussed in the previous chapter, it is often infeasible to compute functions `t2BDDvec` and `f2BDD` precisely. This can be caused either by a large size of some intermediate BDDs or by a large size of the resulting BDD itself. However, even an imprecise result can sometimes be sufficient to decide satisfiability of the input formula as was illustrated in the introduction to this chapter. In this section, we describe general notions of a *term abstract domain*, which captures an imprecise computation of `t2BDDvec`, and a *formula abstract domain*, which captures an imprecise computation of `f2BDD`. In the following definitions, we denote the set of all bit-vector terms of the sort $[n]$ for some n as $\mathcal{T}$ and the set of all bit-vector formulas as $\mathcal{F}$. Recall that $\mathcal{BV}$ is the set of all bit-vectors.

A term abstract domain defines a set of abstract objects A, a function α mapping terms to these abstract objects, and an evaluation function $[\![\_]\!]^A_\_$, which assigns to each abstract object $a \in A$ and a variable assignment μ the set $[\![a]\!]^A_\mu$ of bit-vectors represented by a.

Definition 6.1 (Term abstract domain)**.** *A* term abstract domain *is defined as a triple* $(A, \alpha, [\![\_]\!]^A_\_)$*, where A is a set of* abstract objects*,* $\alpha \colon \mathcal{T} \to A$ *is an* abstraction function*, and* $[\![\_]\!]^A_\_ \colon A \times \mathcal{BV}^{vars} \to 2^{\mathcal{BV}}$ *is an* abstract evaluation function.

As an example, consider the *precise* BDD *term abstract domain*, in which the corresponding vector of precise BDDs is assigned to each term. In particular, the precise BDD term abstract domain is the triple

$$(\texttt{BDDvec}, \texttt{t2BDDvec}, [\![\_]\!]^{\texttt{BDDvec}}_\_),$$

where the value $[\![\overline{a}]\!]^{\texttt{BDDvec}}_\mu$ is the singleton set $\{bv\}$ such that bv is the result of evaluation of vector of BDDs $\overline{a}$ in the assignment μ, i.e., $bv = [\![\overline{a}]\!]_\mu$. Note that this abstract domain serves only as an artificial example as it does not bring any real abstraction. Nevertheless, it enjoys two interesting properties: for each term and assignment, the corresponding abstract object represents the correct result and it does not represent any incorrect result. These properties are called *completeness* and *soundness*, respectively.

Definition 6.2 (Term abstract domain completeness and soundness)**.** *A term abstract domain* $(A, \alpha, [\![\_]\!]^A_\_)$ *is* complete *if each term $t \in \mathcal{T}$ and each assignment μ satisfy* $[\![t]\!]_\mu \in [\![\alpha(t)]\!]^A_\mu$*. Conversely, it is* sound *if each t and μ satisfy* $[\![\alpha(t)]\!]^A_\mu \subseteq \{[\![t]\!]_\mu\}$.

Similarly to the term abstract domain, the *formula abstract domain* defines a set of abstract objects A, a function α mapping *formulas* to these abstract objects, and an evaluation function $[\![\_]\!]^A_\_$, which assigns to each abstract object a and a variable assignment μ the set $[\![a]\!]^A_\mu \subseteq \{0, 1\}$ of truth values associated to a.

Definition 6.3 (Formula abstract domain)**.** *A* formula abstract domain *is a triple* $(A, \alpha, [\![\_]\!]^A_\_)$*, where A is an arbitrary set of* abstract objects*,* $\alpha \colon \mathcal{F} \to A$ *is an* abstraction function*, and* $[\![\_]\!]^A_\_ \colon A \times \mathcal{BV}^{vars} \to 2^{\{0,1\}}$ *is an* abstract evaluation function.

Definition 6.4 (Formula abstract domain completeness and soundness)**.** *A formula abstract domain $(A, \alpha, [\![\_]\!]^A_\_)$ is* complete *if each formula $\varphi \in \mathcal{F}$ and each assignment μ satisfy $[\![\varphi]\!]_\mu \in [\![\alpha(\varphi)]\!]^A_\mu$. Conversely, it is* sound *if each φ and μ satisfy $[\![\alpha(\varphi)]\!]^A_\mu \subseteq \{[\![\varphi]\!]_\mu\}$.*

As in the case of terms, the precise computation of the BDD corresponding to a formula yields a *precise BDD formula abstract domain*, which is complete and sound. The precise BDD formula abstract domain is defined as a triple $(\mathsf{BDD}, \mathtt{f2BDD}, [\![\_]\!]^{\mathsf{BDD}}_\_)$, where $[\![a]\!]^{\mathsf{BDD}}_\mu$ is the singleton set $\{b\}$, where b is the result of evaluation of the BDD a in the assignment μ, i.e., $b = [\![a]\!]_\mu$.

In the following, we weaken the precise term and formula BDD abstract domains by dropping the requirement of soundness, while still retaining the requirement of completeness. As the following theorem demonstrates, such an abstract domain can still be used for deciding satisfiability of the input formula.

Theorem 6.1. *Let φ be a formula and $(A, \alpha, [\![\_]\!]^A_\_)$ be a complete formula abstract domain. If there exists an assignment μ such that $[\![\alpha(\varphi)]\!]^A_\mu = \{1\}$, the formula φ is satisfiable. On the other hand, if all assignments μ satisfy $[\![\alpha(\varphi)]\!]^A_\mu = \{0\}$, the formula is unsatisfiable.*

Proof. Suppose that there is an assignment μ such that $[\![\alpha(\varphi)]\!]^A_\mu = \{1\}$. Since the abstract domain is complete, we know that $[\![\varphi]\!]_\mu \in \{1\}$. Therefore $[\![\varphi]\!]_\mu = 1$ and φ is indeed satisfiable.

For the second claim suppose that all assignments μ satisfy $[\![\alpha(\varphi)]\!]^A_\mu = \{0\}$. Again, from the completeness we know that $[\![\varphi]\!]_\mu \in \{0\}$ for all assignments μ. Therefore $[\![\varphi]\!]_\mu = 0$ for any assignment μ and φ is indeed unsatisfiable. □

6.2 TRUNCATING FORMULA AND TERM ABSTRACT DOMAINS

This section describes a term abstract domain and a corresponding formula abstract domain that allow *truncating* results of bit-vector operations, i.e., computing only several bits from the result of arithmetic bit-vector operations.

In this whole section, we suppose that all formulas are in *negation normal form*. We also treat the negated subformulas $\neg(t_1 \leq_u t_2)$, $\neg(t_1 \leq_s t_2)$, $\neg(t_1 <_u t_2)$, and $\neg(t_1 <_s t_2)$ as the equivalent positive formulas $t_2 <_u t_1$, $t_2 <_s t_1$, $t_2 \leq_u t_1$, and $t_2 \leq_s t_1$, respectively.

6.2.1 *Truncating Term Abstract Domain*

We introduce the *truncating term abstract domain* first. It is a complete but unsound term abstract domain, in which the terms are represented by vectors whose elements are BDDs, as in the precise term abstract domain, or *do-not-know values*. The do-not-know value, denoted as ?, represents an unknown value of the corresponding bit – it can be both 0 and 1.

For example, Figure 6.1 shows the result of computing only the least significant bit of the sum of two bit-vectors $x_2x_1x_0 + y_2y_1y_0$ (compare to Figure 2.3).

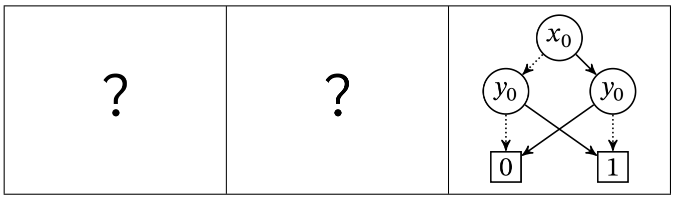


Figure 6.1: Truncated result of addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of size 3.

The value of this abstract object under the assignment $\{x \mapsto 001, y \mapsto 100\}$ is the set $\{001, 011, 101, 111\}$, since only the value of the least significant bit is computed precisely.

Formally, the truncating term abstract domain is a triple

$$(\mathtt{tBDDvec}, \mathtt{t2tBDDvec}, [\![\_]\!]^{\mathtt{tBDDvec}}_{\_}),$$

where the set of abstract elements consists of vectors of BDDs and ? elements

$$\mathtt{tBDDvec} = \left\{(b_i)_{0\leq i<k} \mid k > 0, b_i \in \mathtt{BDD} \cup \{?\} \text{ for all } 0 \leq i < k\right\}$$

and the abstract evaluation function assigns to each $\overline{b} = (b_i)_{0\leq i<k} \in \mathtt{tBDDvec}$ and an assignment μ the following set of bit-vector values

$$[\![\overline{b}]\!]^{\mathtt{tBDDvec}}_{\mu} = \left\{(v_i)_{0\leq i<k} \mid \text{if } b_i = ? \text{ then } v_i \in \{0,1\} \text{ else } v_i = [\![b_i]\!]_{\mu}, 0 \leq i < k\right\}.$$

The function `t2tBDDvec` can be implemented in multiple ways, including the following two:

1. the number of precisely computed bits is fixed and the remaining bits are set to ?,
2. the limit on the number of BDD nodes is specified and after reaching it, the remaining bits are set to ?.

In the following, we focus purely on the second option as our preliminary evaluation has shown that it outperforms the first one. Furthermore, it is easy to derive the implementation of the first option based on the description of the second option.

We suppose that the limit on the number of BDD nodes is fixed for the given domain. Note that our satisfiability deciding procedure can use multiple abstract domains parametrized by the BDD node limit.

The function `t2tBDDvec` is computed recursively on the input term. The base case for the variables and numerals is the same as in the precise function `t2BDDvec` (illustrated in Figure 2.4). The computation for bit-vector operations differs from `t2BDDvec` in two important aspects:

- The operations have to work correctly with ? elements. To achieve this, we modify the BDD operations &, |, xor, and `ite`, which are the building blocks of the `t2BDDvec` computation. The handling of ? in the modified operations is similar to the definition of logical connectives in the three-valued logic and to the way bit-masks are computed in the SMT solver mcBV [ZWR16]. The modified BDD operations $\&_t$, $|_t$, xor_t, and $\mathtt{ite}_t$ are computed as follows:

$$a \ \&_t \ b = \begin{cases} \boxed{0}, & \textit{if } a = \boxed{0} \textit{ or } b = \boxed{0}, \\ a \,\&\, b, & \textit{if } a, b \notin \{\boxed{0}, ?\}, \\ ?, & \textit{otherwise}, \end{cases}$$

$$a \ |_t \ b = \begin{cases} \boxed{1}, & \textit{if } a = \boxed{1} \textit{ or } b = \boxed{1}, \\ a \mid b, & \textit{if } a, b \notin \{\boxed{1}, ?\}, \\ ?, & \textit{otherwise}, \end{cases}$$

$$a \ \mathrm{xor}_t \ b = \begin{cases} a \ \mathrm{xor} \ b, & \textit{if } a, b \neq ?, \\ ?, & \textit{otherwise}, \end{cases}$$

$$\mathtt{ite}_t(a, b, c) = \begin{cases} b, & \textit{if } a = \boxed{1} \textit{ or } b = c, \\ c, & \textit{if } a = \boxed{0}, \\ \mathtt{ite}(a, b, c), & \textit{if } a \notin \{\boxed{0}, \boxed{1}, ?\},\ b, c \neq ? \textit{ and } b \neq c, \\ ?, & \textit{otherwise}. \end{cases}$$

Note that $? \ \mathrm{xor}_t \ ?$ is not $\boxed{0}$ as each ? can represent a different value.

- Implementation of operations has to consider the given limit on the number of BDD nodes and set the bits that have not been computed precisely to ? after the limit has been reached. Listing 6.1 shows the algorithms computing *truncated addition* and *truncated multiplication*, which are the standard addition and multiplication functions extended with the node limit (compare to the original functions given in Listing 2.1). Both algorithms use the function `bddNodes`, which returns the total number of BDD nodes in a given vector of BDDs. In the implementation of truncated addition, the precise results are computed until the node limit is reached; all more significant bits after that are replaced by ?. The algorithm for multiplication is modified in a similar way: once the limit is reached, all bits of the result that might be modified in the rest of the precise algorithm are set to ?.

  The implementations of other truncated operations are similar. However, they may differ in the order in which the precise bits are produced: during computation of addition and multiplication, the first precisely computed bits are the least significant ones; during computation of division, the first precisely computed bits are the most significant ones. Therefore, if the computation of addition or multiplication reaches the

```
bvec_add_trunc($\overline{a}$, $\overline{b}$, limit)
{
  $\overline{result}$ ← $(\boxed{0}, \boxed{0}, \ldots, \boxed{0})$ with the bit-width $k$;
  $carry$ ← $\boxed{0}$;

  for $i$ from 0 to $k$ - 1 {
    if (bddNodes($\overline{result}$) > limit) {
      $result_i$ ← ?;
    } else {
      $result_i$ ← $a_i$ xor$_t$ $b_i$ xor$_t$ $carry$;
      $carry$ ← $(a_i$ &$_t$ $b_i)$ |$_t$ $(carry$ &$_t$ $(a_i$ |$_t$ $b_i))$;
    }
  }
  return $\overline{result}$;
}

bvec_mul_trunc($\overline{a}$, $\overline{b}$, limit)
{
  $\overline{result}$ ← $(\boxed{0}, \boxed{0}, \ldots, \boxed{0})$ with the bit-width$k$;

  for $i$ from 0 to $k$ - 1 {
    $\overline{added}$ ← bvec_add($\overline{result}, \overline{a}$);

    for $j$ from $i$ to $k - 1$ {
      $result_j$ ← ite$_t(b_i, added_j, result_j)$;
      if (bddNodes($\overline{result}$) > limit) {
        //too many nodes; set the remaining bits to DNK
        for $m$ from $i + 1$ to $k - 1$ {
          $result_m$ ← ?;
        }
        return $\overline{result}$;
      }
    }

    $\overline{a}$ ← bvec_shl($\overline{a}$, 1);
  }
  return $\overline{result}$;
}
```

Listing 6.1: Functions bvec_add_trunc and bvec_mul_trunc implementing *truncated addition* and *truncated multiplication* of two tBDDvecs $\overline{a} = (a_i)_{0 \le i < k}$ and $\overline{b} = (b_i)_{0 \le i < k}$.

BDD node limit, remaining more significant bits are set to ?, while for division less significant bits are set to ?.

For each assignment, the set of values represented by the result of the function `t2tBDDvec` always contains the precise result of the given term because the function `t2tBDDvec` can only make precise values imprecise by using ? elements. The truncating term abstract domain is therefore complete, as is stated by the following theorem. However, it is not sound, as the abstract object can describe also incorrect results.

Theorem 6.2. *The truncating term abstract domain is complete. Moreover, each element* `t2tBDDvec`$(t)_i$ *is either equal to the* BDD `t2BDDvec`$(t)_i$ *or it is* ?.

6.2.2 *Truncating Formula Abstract Domain*

We now define a formula abstract domain that uses results of truncated bit-vector operations. The abstract elements of this abstract domain are BDD pairs (b_{must}, b_{may}). Intuitively, b_{must} determines the assignments that satisfy the formula for all possible values of ? elements and b_{may} determines the assignments that satisfy the formula for some values of ? elements. In other words, b_{must} represents a subset of all formula models while b_{may} represents a superset of all formula models.

Formally, the *truncating formula abstract domain* is a triple

$$(\texttt{BDDpair}, \texttt{f2BDDpair}, [\![\_]\!]_{\_}^{\texttt{BDDpair}}),$$

where $\texttt{BDDpair} = \texttt{BDD} \times \texttt{BDD}$ and the evaluation function assigns to each pair $(b_{must}, b_{may}) \in \texttt{BDDpair}$ and an assignment μ the set of Boolean values

$$[\![(b_{must}, b_{may})]\!]_{\mu}^{\texttt{BDDpair}} = \left\{v \in \{0,1\} \mid [\![b_{must}]\!]_{\mu} \implies v \text{ and } v \implies [\![b_{may}]\!]_{\mu}\right\}.$$

Observe that $[\![(b_{must}, b_{may})]\!]_{\mu}^{\texttt{BDDpair}}$ is $\{0\}$ when $[\![b_{must}]\!]_{\mu} = [\![b_{may}]\!]_{\mu} = 0$, it is $\{1\}$ when $[\![b_{must}]\!]_{\mu} = [\![b_{may}]\!]_{\mu} = 1$, and it is $\{0,1\}$ when $[\![b_{must}]\!]_{\mu} = 0$, $[\![b_{may}]\!]_{\mu} = 1$. The result would be $\emptyset$ in the remaining case $[\![b_{must}]\!]_{\mu} = 1$, $[\![b_{may}]\!]_{\mu} = 0$, but this situation never happens for the BDD pairs produced by `f2BDDpair`.

The function `f2BDDpair`(φ) is defined recursively as follows.

1. The formula φ is an atomic subformula or its negation, i.e., $\varphi \equiv t_1 \bowtie t_2$ for $\bowtie \in \{=, \neq, \leq_u, <_u, \leq_s, <_s\}$. The function `f2BDDpair` computes the pair (b_{must}, b_{may}) from `t2tBDDvec`(t_1) and `t2tBDDvec`(t_2) using modified algorithms for the corresponding predicates on the vectors of standard BDDs. For example, Listing 6.2 shows an algorithm computing *truncated equality* (compare to the original function for equality of vectors of standard BDDs presented in Listing 2.2). In this algorithm, the value b_{must} becomes $\boxed{0}$ if there is ? in any of the input vectors, because then the arguments may differ for some value of the ?. On the other hand, the value b_{may} is the conjunction of equalities of all pairs of corresponding bits that both have a known value. In particular, construction of b_{may}

ignores the pairs of bits containing some ? as the equality may hold for these bits. Listing 6.2 also shows the algorithms computing *truncated disequality* and *truncated unsigned inequality*. The algorithms for other predicates are similar.

2. The formula φ has the form $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$. Let (b^1_{must}, b^1_{may}) be the result of `f2BDDpair`(φ_1) and (b^2_{must}, b^2_{may}) be the result of `f2BDDpair`(φ_2). Then we define

$$\texttt{f2BDDpair}(\varphi_1 \wedge \varphi_2) = ((b^1_{must} \mathbin{\&} b^2_{must}),\ (b^1_{may} \mathbin{\&} b^2_{may})),$$
$$\texttt{f2BDDpair}(\varphi_1 \vee \varphi_2) = ((b^1_{must} \mid b^2_{must}),\ (b^1_{may} \mid b^2_{may})).$$

3. The formula φ has the form $\forall x\,(\varphi_1)$ or $\exists x\,(\varphi_1)$. Let (b^1_{must}, b^1_{may}) be the result of `f2BDDpair`(φ_1). Then we define

$$\texttt{f2BDDpair}(\forall x\,(\varphi_1)) = (\mathit{bdd\_forall}(x, b^1_{must}),\ \mathit{bdd\_forall}(x, b^1_{may})),$$
$$\texttt{f2BDDpair}(\exists x\,(\varphi_1)) = (\mathit{bdd\_exists}(x, b^1_{must}),\ \mathit{bdd\_exists}(x, b^1_{may})),$$

where the function $\mathit{bdd\_forall}(x, \_)$ eliminates all the BDD variables that form the bit-vector variable x universally and $\mathit{bdd\_exists}(x, \_)$ eliminates them existentially as explained in Section 2.4.1.

Example 6.1. *Let t, r, s, and u be bit-vector terms, for which we have computed only the least significant bit as computation of the other bits was infeasible. Formally,*

$$\texttt{t2tBDDvec}(t) = (?, \dots, ?, b_t), \qquad \texttt{t2tBDDvec}(r) = (?, \dots, ?, b_r),$$
$$\texttt{t2tBDDvec}(s) = (?, \dots, ?, b_s), \qquad \texttt{t2tBDDvec}(u) = (?, \dots, ?, b_u),$$

where b_t, b_r, b_s, and b_u are BDDs.

Consider the formula $t = r$. The function `f2BDDpair` *applied on this formula returns the pair $(\boxed{0}, b_t \leftrightarrow b_r)$. The pair says that an assignment may satisfy the formula $t = r$ only if it satisfies $b_t \leftrightarrow b_r$. Therefore, if $t = r$ is put in conjunction with another formula implying that $b_t \leftrightarrow b_r$ is equal to $\boxed{0}$, the whole conjunction can be decided as unsatisfiable.*

Consider the formula $s \neq u$. The function `f2BDDpair` *now produces the pair $(b_s \operatorname{xor} b_u, \boxed{1})$. Intuitively, if an assignment satisfies $b_s \operatorname{xor} b_u$, it also satisfies the formula $s \neq u$, regardless the values of the remaining bits of s and u.*

Further, consider the formula $t = r \wedge s \neq u$. The result of `f2BDDpair` *applied to this formula is computed as $(\boxed{0} \mathbin{\&} (b_s \operatorname{xor} b_u),\ (b_t \leftrightarrow b_r) \mathbin{\&} \boxed{1})$, which can be simplified to $(\boxed{0}, b_t \leftrightarrow b_r)$.*

Finally, consider the formula $t = r \vee s \neq u$. The result of `f2BDDpair` *applied to this formula is computed as $(\boxed{0} \mid (b_s \operatorname{xor} b_u),\ (b_t \leftrightarrow b_r) \mid \boxed{1})$, which is clearly equivalent to $(b_s \operatorname{xor} b_u, \boxed{1})$.*

```
bvec_eq_trunc($\overline{a}$, $\overline{b}$)
{
  $result_{must}$ ← $\boxed{1}$;
  $result_{may}$ ← $\boxed{1}$;
  for $i$ from 0 to $k$ - 1 {
    if ($a_i$ == ? or $b_i$ == ?) {
      $result_{must}$ ← $\boxed{0}$;
    } else {
      $result_{must}$ ← $result_{must}$ & $(a_i \leftrightarrow b_i)$;
      $result_{may}$ ← $result_{may}$ & $(a_i \leftrightarrow b_i)$;
    }
  }
  return ($result_{must}$, $result_{may}$);
}

bvec_neq_trunc($\overline{a}$, $\overline{b}$)
{
  $result_{must}$ ← $\boxed{0}$;
  $result_{may}$ ← $\boxed{0}$;
  for $i$ from 0 to $k$ - 1 {
    if ($a_i$ == ? or $b_i$ == ?) {
      $result_{may}$ ← $\boxed{1}$;
    } else {
      $result_{must}$ ← $result_{must} \mid (a_i \text{ xor } b_i)$;
      $result_{may}$ ← $result_{may} \mid (a_i \text{ xor } b_i)$;
    }
  }
  return ($result_{must}$, $result_{may}$);
}

bvec_leq_trunc($\overline{a}$, $\overline{b}$)
{
  $result_{must}$ ← $\boxed{1}$;
  $result_{may}$ ← $\boxed{1}$;
  for $i$ from 0 to $k$ - 1 {
    if ($a_i$ == ? or $b_i$ == ?) {
      $result_{must}$ ← $\boxed{0}$;
      $result_{may}$ ← $\boxed{1}$;
    } else {
      $result_{must}$ ← $(!a_i$ & $b_i) \mid (result_{must}$ & $(a_i \leftrightarrow b_i))$
      $result_{may}$ ← $(!a_i$ & $b_i) \mid (result_{may}$ & $(a_i \leftrightarrow b_i))$
    }
  }
  return ($result_{must}$, $result_{may}$);
}
```

Listing 6.2: Algorithms bvec_eq_trunc, bvec_neq_trunc, and bvec_leq_trunc implementing the functions for *truncated equality*, *truncated disequality*, and *truncated unsigned inequality* of two tBDDvecs $\overline{a} = (a_i)_{0 \le i < k}$ and $\overline{b} = (b_i)_{0 \le i < k}$.

Similarly to the truncating term abstract domain, the truncating formula abstract domain is also complete, as the following theorem shows.

Theorem 6.3. *The truncating formula abstract domain is complete.*

Proof. We show that if `f2BDDpair`$(\varphi) = (b_{must}, b_{may})$ for a formula φ, then the following must hold for each assignment μ:

$$[\![b_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu \quad \text{and} \quad [\![\varphi]\!]_\mu \implies [\![b_{may}]\!]_\mu.$$

This implies completeness of the abstract domain, since the claim directly implies that $[\![\varphi]\!]_\mu \in [\![\texttt{f2BDDpair}(\varphi)]\!]_\mu^{\texttt{BDDpair}}$ holds for each assignment μ.

We show that the aforementioned claim holds by induction on the structure of the formula:

1. The formula φ is an atomic subformula or its negation, i.e., $\varphi \equiv t_1 \bowtie t_2$ for $\bowtie \in \{=, \neq, \leq_u, <_u, \leq_s, <_s\}$. From the construction of the truncating *term* abstract domain, we know that for both vectors of BDDs that are given as arguments, each bit is either the precise BDD or ? (see Theorem 6.2).

   We first consider the case for $\varphi \equiv t_1 = t_2$. If no input BDDs are ?, the claim trivially holds. Otherwise, if some input BDDs are ?, then the BDD $result_{must}$ returned by algorithm `bvec_eq_trunc` is $\boxed{0}$, and thus definitely $[\![result_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu$. On the other hand, the returned BDD $result_{may}$ misses some of the conjuncts compared to the precise result. Therefore the precise result implies $result_{may}$, and thus $[\![\varphi]\!]_\mu \implies [\![result_{may}]\!]_\mu$. The situation is dual for $\varphi \equiv t_1 \neq t_2$ and `bvec_neq_trunc`.

   If $\bowtie = \leq_u$ and the arguments contain some ? bits, observe that the algorithm `bvec_leq_trunc` may repeatedly set $result_{must}$ to $\boxed{0}$ and $result_{may}$ to $\boxed{1}$ instead of the precise result. Consider the last such step. All the following iterations modify both $result_{must}$ and $result_{may}$ by applying the operations & and | with the correct values. Since both & and | are monotonous, it follows that $[\![result_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu$ and $[\![\varphi]\!]_\mu \implies [\![result_{may}]\!]_\mu$.

   The argumentation for the cases $t_1 <_u t_2$, $t_1 \leq_s t_2$, and $t_1 <_s t_2$ is analogous.

2. The formula φ has the form $\varphi_1 \land \varphi_2$. From the definition of `f2BDDpair`, we know that

   $$\texttt{f2BDDpair}(\varphi_1 \land \varphi_2) = ((b^1_{must} \,\&\, b^2_{must}),\ (b^1_{may} \,\&\, b^2_{may})),$$

   where (b^1_{must}, b^1_{may}) is the result of `f2BDDpair`(φ_1) and (b^2_{must}, b^2_{may}) is the result of `f2BDDpair`(φ_2). Let μ be an arbitrary assignment. We know from the induction hypothesis that $[\![b^1_{must}]\!]_\mu$ implies $[\![\varphi_1]\!]_\mu$ and $[\![b^2_{must}]\!]_\mu$ implies $[\![\varphi_2]\!]_\mu$. Because the operation computing conjunction of BDDs is correct, it follows that $[\![b^1_{must} \,\&\, b^2_{must}]\!]_\mu \equiv [\![b^1_{must}]\!]_\mu \land [\![b^2_{must}]\!]_\mu$. Therefore we know that $[\![b^1_{must} \,\&\, b^2_{must}]\!]_\mu \implies [\![\varphi_1]\!]_\mu$ and $[\![b^1_{must} \,\&\, b^2_{must}]\!]_\mu \implies$

$[\![\varphi_2]\!]_\mu$. From the definition of conjunction and the evaluation function $[\![\_]\!]_\mu$, we conclude

$$[\![b^1_{must} \mathbin{\&} b^2_{must}]\!]_\mu \implies [\![\varphi_1]\!]_\mu \wedge [\![\varphi_2]\!]_\mu \equiv [\![\varphi_1 \wedge \varphi_2]\!]_\mu.$$

We also know from the induction hypothesis that $[\![\varphi_1]\!]_\mu \implies [\![b^1_{may}]\!]_\mu$ and $[\![\varphi_2]\!]_\mu \implies [\![b^2_{may}]\!]_\mu$. From the definition of the evaluation function thus $[\![\varphi_1 \wedge \varphi_2]\!]_\mu \equiv [\![\varphi_1]\!]_\mu \wedge [\![\varphi_2]\!]_\mu$ and therefore $[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^1_{may}]\!]_\mu$ and $[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^2_{may}]\!]_\mu$. Therefore from the correctness of conjunction of BDDs, we can conclude that

$$[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^1_{may} \mathbin{\&} b^2_{may}]\!]_\mu.$$

The case when the formula φ has the form $\varphi_1 \vee \varphi_2$ is analogous to the previous one.

3. The formula φ has the form $\forall x\,(\varphi_1)$. From the definition of `f2BDDpair` we know that

$$\texttt{f2BDDpair}\,(\forall x\,(\varphi_1)) = (\mathit{bdd\_forall}(x, b^1_{must}),\ \mathit{bdd\_forall}(x, b^1_{may})),$$

where (b^1_{must}, b^1_{may}) is the result of `f2BDDpair`(φ_1). Let μ be an arbitrary assignment. We want to show $[\![\mathit{bdd\_forall}(x, b^1_{must})]\!]_\mu \implies [\![\forall x\,(\varphi_1)]\!]_\mu$ and $[\![\forall x\,(\varphi_1)]\!]_\mu \implies [\![\mathit{bdd\_forall}(x, b^1_{may})]\!]_\mu$.

For the first implication, assume that $[\![\mathit{bdd\_forall}(x, b^1_{must})]\!]_\mu = 1$ and v is an arbitrary bit-vector of the same bit-width as x. We want to show that $[\![\varphi_1]\!]_{\mu[x\mapsto v]} = 1$. From the assumption we know that $[\![b^1_{must}]\!]_{\mu[x\mapsto v]} = 1$ since the operation *bdd_forall* is correct. From the induction hypothesis we know that $[\![b^1_{must}]\!]_{\mu[x\mapsto v]}$ implies $[\![\varphi_1]\!]_{\mu[x\mapsto v]}$, and thus $[\![\varphi_1]\!]_{\mu[x\mapsto v]} = 1$.

For the second implication, assume that $[\![\forall x\,(\varphi_1)]\!]_\mu = 1$. We will show that $[\![\mathit{bdd\_forall}(x, b^1_{may})]\!]_\mu = 1$. Because the operation *bdd_forall* is correct, it is enough to show that $[\![b^1_{may}]\!]_{\mu[x\mapsto v]} = 1$ for an arbitrary bit-vector v of the same bit-width as x. From the assumption we know that $[\![\varphi_1]\!]_{\mu[x\mapsto v]} = 1$. From the induction hypothesis we know that $[\![\varphi_1]\!]_{\mu[x\mapsto v]}$ implies $[\![b^1_{may}]\!]_{\mu[x\mapsto v]}$ and thus, indeed, $[\![b^1_{may}]\!]_{\mu[x\mapsto v]} = 1$.

The case when the formula φ has the form $\exists x\,(\varphi_1)$ is again analogous to the previous one. □

Since the truncating formula abstract domain is complete, Theorem 6.1 can be used to check satisfiability of a given formula φ. Assume that the value `f2BDDpair`(φ) is (b_{must}, b_{may}). Then if b_{must} is not $\boxed{0}$, the formula φ is satisfiable. Furthermore, if b_{may} is $\boxed{0}$, the formula φ is unsatisfiable.

This satisfiability check solves the formulas mentioned in the beginning of this chapter as the motivation for the described approach. For all three of the formulas, b_{may} is $\boxed{0}$ after computing at most 5 bits of the multiplication. Thus the formulas can be decided as unsatisfiable.

```
solve_underapproximations($\varphi$)
{
  effBW ← initialEffBW;
  nodeLimit ← initialNodeLimit;
  while (true) {
    $\varphi_u \leftarrow \underline{\varphi}_e$;
    ($b_{must}$, $b_{may}$) ← f2BDDpair($\varphi_u$, nodeLimit);
    if ($b_{must}$ != $\boxed{0}$) return SAT;
    if ($b_{may}$ == $\boxed{0}$ and $\varphi$ == $\varphi_u$) return UNSAT;
    if ($b_{must}$ != $b_{may}$) {
      nodeLimit ← increaseNodeLimit(nodeLimit);
    }
    if ($b_{must}$ == $b_{may}$ and $\varphi$ != $\varphi_u$) {
      effBW ← increaseEffBW(effBW);
    }
  }
}
```

Listing 6.3: Algorithm that combines abstractions of bit-vector operations and formula underapproximation.

6.3 OPERATION ABSTRACTIONS WITH FORMULA APPROXIMATIONS

We now show how to combine the proposed operation abstractions with the formula approximations, which were introduced in the previous chapter. Recall that the formula approximations are performed on formulas by reducing the *effective bit-width* of selected variables by fixing some of their bits to chosen values. The underapproximations are obtained by decreasing effective bit-widths of all existentially quantified variables and the overapproximations are obtained by decreasing effective bit-widths of all universally quantified variables. The introduced approach tries to solve the input formula by solving the original formula, underapproximations of the formula, and overapproximations of the formula in parallel. We show how to integrate the proposed operation abstractions into the functions for solving underapproximations and overapproximations. The function solving the original formula can be adjusted to use operation abstractions as well, but according to our experiments, the tool performs better if we keep this function unchanged.

Listing 6.3 shows the modified pseudocode for the function that solves the formula by using its underapproximations. The algorithm starts with the small initial values of the effective bit-width `effBW` of existential variables and the limit `nodeLimit` on the number of BDD nodes in the results of arithmetic operations. It repeatedly tries to solve the input formula and if the result is not determined, either the effective bit-width or the node limit is increased:

- If the operation abstractions caused an imprecision, the node limit is increased. Observe that the source of potential imprecision can be determined by checking equality of b_{must} and b_{may}.

- If the BDD pair returned by `f2BDDpair` was precise, but the reduced effective bit-width could have caused imprecision, the effective bit-width is increased.

The operation `solve_overapproximations` can be implemented analogously.

6.4 FURTHER EXTENSIONS

The described approach can be extended in several ways. For example, the returned BDD b_{may} can be used even in cases in which the abstraction cannot be used to decide the satisfiability of the input formula. We describe two such usages. First of these is checking the potential models described by b_{may}, which is explained in the following Subsection 6.4.1. Subsection 6.4.2 then shows that b_{may} may be also used to identify the bits whose values are implied by the input formula.

Further, Subsection 6.4.3 introduces formula modifications that add new variables for results of multiplications and divisions and their respective congruences, which can further amplify the positive effect of the truncating abstractions.

6.4.1 *Checking Possible Models*

In the described method of checking satisfiability using abstractions, the result is unknown if the BDD b_{must} is $\boxed{0}$ and b_{may} is not $\boxed{0}$. However, even in this case, b_{may} can sometimes be used to decide the satisfiability of the formula: one can extract a model from b_{may}, substitute it into the input formula, and check satisfiability of the resulting formula φ_{subst}. If the formula φ_{subst} is satisfiable, the input formula is also satisfiable. Only if the formula φ_{subst} is not satisfiable, the result is unknown.

However, checking models of quantified formulas is more involved. The problem is that the potential model contains values only for free variables and the top-level existential variables, and therefore the formula φ_{subst} can still contain not only universal quantifiers, but also existential quantifiers that were in scope of some universal quantifier. Such a formula cannot be directly evaluated to yield 1 or 0 and has to be decided by a new instance of the solver for quantified formulas. This may be problematic because the satisfiability check of φ_{subst} can be expensive and we want to ensure that the model checking finishes quickly. Therefore, we propose computing not the precise BDD for φ_{subst}, but only its b'_{must} with the lowest possible precision. Note that this computation will usually finish in a short time. If the resulting b'_{must} is not $\boxed{0}$, the formula φ_{subst} is satisfiable, and thus also the input formula is satisfiable. In the opposite case, b'_{must} is $\boxed{0}$ and the potential model is discarded, because it is either not a model of the input formula or could not be validated quickly.

Note that without operation abstractions, there is no guarantee that the model checking would finish quickly. For example, the formula φ_{subst} might be quantifier-free and thus the formula underapproximation introduced in the previous chapter cannot speed the model checking up, because there are no universally quantified variables whose effective bit-width could be reduced. For this reason, model checking was not proposed in our original approach.

A dual approach can be used for closed formulas that contain universally quantified variables on the top-level of a formula. If a BDD b_{must} for such formula is $\boxed{0}$, one can identify a potential countermodel, i.e., an assignment of values to the top-level universal variables that makes the formula unsatisfiable. Such a potential countermodel can then be checked against the original formula.

Since no countermodel can be computed directly from $b_{must} = \boxed{0}$, the proposed counter-model checking can be implemented by negating all closed input formulas that have an universal top-level quantifier. This reduces the situation to the one described in the beginning of this subsection: b_{may} is computed instead of b_{must}; if it is not $\boxed{0}$, one of its models is extracted. Since the input formula φ was negated, this model now corresponds to the assignment to the top-level universally quantified variables of the original formula φ. This potential countermodel is then substituted into the original formula φ and the satisfiability of the resulting formula φ_{subst} is checked by computing the corresponding b'_{may}. If b'_{may} is $\boxed{0}$, the formula φ_{subst} is unsatisfiable an thus the input formula is also unsatisfiable. If b'_{may} is not $\boxed{0}$, the result is unknown.

6.4.2 *Learning From Overapproximations*

The BDD b_{may} can be used not only for generating potential models, but also for identifying values of some bits that are necessary to satisfy the formula. Since b_{may} represents a superset of all the satisfying assignments, if the value of some bit b is 1 in all of these assignments, this bit must be 1 if the input formula should be satisfied. This also works analogously if the value of a bit is 0 in all the represented assignments. After identifying all such bits, we can replace them by their implied values and try solving the resulting formula with an increased node limit. In practice, these bits and their implied values can by identified by the functionality provided by BDD packages. For example, the package CUDD [Som15] offers the corresponding function `FindEssential()`.

As an example, consider the formula $x \leq_u 30 \;\wedge\; x \times y = 0$ where x, y are 32-bit variables. The aforementioned procedure can identify that the most significant 27 bits of x must be zero. However, consider the following formula:

$$x \leq_s 4 \;\;\wedge\;\; x \geq_s -4 \;\;\wedge\;\; x \times y = 0.$$

Although the most significant 29 bits of x must be either all 0 (if x is positive) or all 1 (if x is negative), the aforementioned procedure cannot identify this, as these bits have more possible values. Therefore, we also propose a procedure that identifies which successive bits of a variable must be equal for the formula to be true. In the previous example, we can thus identify that the most significant 29 bits of x must be all the same and therefore we can replace them all by a single BDD variable and start the solving again with an increased node limit. The pseudocode for the procedure that identifies equalities implied by b_{may} is presented in Listing 6.4. The idea of this procedure is straightforward:

```
get_implied_eqns($\varphi$, $b_{may}$)
{
  impliedEqualities ← ∅;
  foreach $\overline{v}$ in $vars(\varphi)$ {
    foreach i from 0 to $bitWidth(\overline{v}) - 2$ {
      if (($b_{may}$ & ($v_i$ xor $v_{i+1}$)) == $\boxed{0}$) {
        impliedEqualities ← impliedEqualities ∪ {($v_i$, $v_{i+1}$)};
      }
    }
  }
  return impliedEqualities;
}
```

Listing 6.4: The algorithm that computes equivalences of the successive bits of variables that are implied by the input formula.

if a conjunction of b_{may} with a xor b is $\boxed{0}$, there is no satisfying assignment of the formula with different values of a and b. And thus the values of a and b must be the same if the formula should be satisfied.

6.4.3 *Adding New Variables and Congruences*

The abstractions by themselves cannot directly solve simple formulas as

$$x \times y \leq_u 2 \land x \times y \geq_u 4. \tag{6.1}$$

Even if the subterms $x \times y$ are computed abstractly, the information that the ? elements in the two vectors representing the two occurrences of $x \times y$ have been the same is lost after computing BDD pairs for $x \times y \leq_u 2$ and $x \times y \geq_u 4$. Therefore, we propose replacing each multiplication and division by a fresh existentially quantified variable of the corresponding sort and add to the modified formula the constraint that specifies the relation of the new variable to the corresponding multiplication or division, respectively. For example, the previous formula is transformed to the equivalent formula

$$\exists m_{x,y}(m_{x,y} \leq_u 2 \land m_{x,y} \geq_u 4 \land m_{x,y} = x \times y).$$

This formula is decided as unsatisfiable even if $x \times y$ is computed with arbitrarily low precision. Although this particular case could be solved by the original approach without operation abstractions by computing precise BDD only for the subformula $m_{x,y} \leq_u 2 \land m_{x,y} \geq_u 4$ (and not for the third conjunct) as this conjunction is already unsatisfiable, the usage of operation abstractions is more general.

A similar problem arises for example in the unsatisfiable formula

$$x \times y \leq_u 2 \land \forall z\,(z \times y \geq_u 4),$$

which logically entails the above-mentioned formula (6.1). This formula cannot be solved even after performing the above-mentioned transformation. The transformation yields the formula

$$\begin{aligned}&\exists m_{x,y}(m_{x,y} \leq_u 2 \ \wedge \\ &\qquad m_{x,y} = x \times y \ \wedge \\ &\qquad \forall z \, \exists m_{z,y} \, (m_{z,y} \geq_u 4 \ \wedge \ m_{z,y} = z \times y)),\end{aligned}$$

which cannot be decided unsatisfiable even by using the abstractions, because the solver cannot infer the relationship between variables $m_{x,y}$ and $m_{z,y}$ without computing the multiplication results precisely. To solve such formula, we propose to add a congruence subformula stating that $(x = z) \rightarrow (m_{x,y} = m_{z,y})$ to the formula. This results in the formula

$$\begin{aligned}&\exists m_{x,y}\Big(m_{x,y} \leq_u 2 \ \wedge \\ &\qquad m_{x,y} = x \times y \ \wedge \\ &\qquad \forall z \, \exists m_{z,y} \, \big(m_{z,y} \geq_u 4 \ \wedge \ m_{z,y} = z \times y \ \wedge \ ((x = z) \rightarrow (m_{x,y} = m_{z,y}))\big)\Big),\end{aligned}$$

which can be decided unsatisfiable using the abstractions. Similarly to the previous transformation, the resulting formula is equivalent to the original one and its unsatisfiability cannot be shown by the original solver without the abstractions, because it is infeasible to compute the precise BDD for the inner quantified subformula.

We now describe the proposed formula modifications precisely. For simplicity, suppose that the input formula is closed. The extension to non-closed formulas is straightforward. The modifications proceed in two steps:

1. In the first step, we introduce constants $m_{x,y}$ for each subterm $x \times y$, where x and y are bit-vector variables. Namely, we recursively traverse the formula and identify all subformulas of the form $Q_1 x \, (\psi)$, where ψ has a subformula of the form $Q_2 y \, (\rho)$ and ρ contains a subterm $x \times y$ or $y \times x$ for $Q_1, Q_2 \in \{\exists, \forall\}$ and bit-vector variables x, y. We replace all such subformulas ρ by

   $$\rho' \ \equiv \ \exists m_{x,y} \, (\rho[(x \times y) \leftarrow m_{x,y}] \ \wedge \ m_{x,y} = x \times y),$$

   where $m_{x,y}$ is a fresh variable of the same sort as x and y, if the found subterm was $x \times y$, and analogously for the case of $y \times x$.

2. In the resulting formula, we add the subformulas expressing the congruences for the variables $m_{x,y}$. We iterate through all the modified subformulas $\rho' \ \equiv \ \exists m_{x,y} \, (\rho[(x \times y) \leftarrow m_{x,y}] \ \wedge \ m_{x,y} = x \times y)$ such that the occurrence of the subformula ρ' is in scope of some newly introduced variable $m_{z,v}$. Based on the syntactic equality of the variables x, y, z, and v, we then perform one of the following modifications:
   - If $x \neq z$ and $y = v$, the body of the quantified subformula ρ' is conjoined with the formula $(x = z) \rightarrow (m_{x,y} = m_{z,y})$.

- If $x = z$ and $y \neq v$, the body of the quantified subformula ρ' is conjoined with the formula $(y = v) \rightarrow (m_{x,y} = m_{x,v})$.
- If $x \neq z$ and $y \neq v$, the body of the quantified subformula ρ' is conjoined with the formula $(x = z \wedge y = v) \rightarrow (m_{x,y} = m_{z,v})$.

The analogous procedure should also be implemented for the results of division of two variables, not only for multiplications. We propose not implementing the transformations for the results of additions, as they were not helpful during in our preliminary experiments.

7 SIMPLIFICATION OF FORMULAS WITH UNCONSTRAINED VARIABLES AND QUANTIFIERS

For most of the modern SMT solvers, preprocessing of the input formula is a crucial step for the efficiency of the solver. Therefore, modern SMT solvers employ hundreds of rewrite rules in order to simplify the input formula [Fra10]. The aim of most of the simplifications is to reduce the size of the input formula and to replace expensive operations by easier ones. One class of these simplification rules focuses on formulas containing unconstrained variables. An unconstrained variable is a variable that occurs only once in the formula and therefore can be set to any suitable value without affecting the rest of the formula. For example, the formula $x + (5 \times y + z) = y \times z$ can be rewritten to an equisatisfiable formula $u = y \times z$ because, regardless of the values of y and z, the term $x + (5 \times y + z)$ can be evaluated to any value of u by choosing a suitable value of x. Such terms, which can be set to an arbitrary value by a well-suited choice of values of unconstrained variables, are called *unconstrained terms*. The principle of simplifications of unconstrained terms is recalled in more detail in Section 7.1. This simplification technique was proposed by Bruttomesso [Bru08] and Brummayer [Bru10], who independently observed that industrial benchmarks often contain non-trivial amount of unconstrained variables. For example, consider SMT queries coming from symbolic execution [Kin76] of a program, where a query is satisfiable if and only if the symbolically executed program path is feasible. There are basically two sources of unconstrained variables in such queries. One source is input variables: an input variable is unconstrained in all queries corresponding to the symbolic execution of a path that reads the input variable at most once. The second source is program variables that are assigned on an executed path, but not read yet. For instance, the execution of an assignment `y:=x+5` leads to a conjunct $y = x + 5$ in the path condition query, where y does not appear anywhere else in the query (unless it is read) and thus it is unconstrained. Such situations are especially frequent when analyzing Static Single Assignment (SSA) code such as LLVM [LA04], which uses many program variables.

In this chapter, we extend the notion of unconstrained terms in several ways:

- In some cases, the definition of unconstrained term is too restrictive by allowing only terms that can evaluate to every possible value by a suitable choice of values of unconstrained variables. For example, Bruttomesso and Brummayer describe the simplification rule that replaces the bit-vector term $c^{[n]} \times x^{[n]}$ by a fresh variable $y^{[n]}$, if $x^{[n]}$ is an unconstrained variable and c is an odd numeral. However, if c is even, the simplification is no longer possible. We describe a less restrictive simplification using *partially constrained terms*, which for example allows replacing the term $6^{[n]} \times x^{[n]}$ by the term $2^{[n]} \times y^{[n]}$; although these two terms can not evaluate to all possible values, they can evaluate to precisely the same set of values.

Partially constrained terms are studied in Section 7.2. This section also shows that several *ad-hoc* simplification rules introduced by Bruttomesso can be seen as instances of simplification of partially constrained terms. Our definition of partially constrained terms allows construction of more similar rules.

- Previously, the simplifications of unconstrained terms were described only on quantifier-free formulas. In Section 7.3, we formalize the conditions under which the simplification of unconstrained terms can be performed on *quantified* formulas.
- Section 7.4 combines techniques from the two preceding sections and describes simplification of *partially constrained terms in quantified formulas*. Furthermore, the resulting technique is combined with additional quantifier-specific simplification rules to allow more efficient and straightforward applications.
- Section 7.5 extends the simplifications of partially constrained terms in quantifier-free formulas by taking into account also the location of the simplified term in the formula. Namely, we take into account whether the term should be unsigned-minimized, unsigned-maximized, signed-minimized, or signed-maximized. We call such terms as *goal unconstrained terms*.
- Section 7.6 extends the goal unconstrained terms to formulas with quantifiers.

We emphasize that the presented approach is not tied to any particular theory. We use the bit-vector theory in many examples as its functions tend to produce unconstrained terms when at least one argument is an unconstrained variable.

7.1 UNCONSTRAINED TERMS IN QUANTIFIER-FREE FORMULAS

This section formalizes known simplifications of quantifier-free formulas containing *unconstrained terms*. Intuitively, a term t is unconstrained in the formula φ if for every assignment to the variables occurring in the term, every possible value of the sort of the term t can be obtained by changing values of only variables that are unconstrained in φ. The idea of simplification is to replace a nontrivial unconstrained term by a fresh variable, which leads to a smaller equisatisfiable formula. For example, consider the formula

$$(x + (3 \times y) = 0 \;\wedge\; y >_u 0)$$

in the bit-vector theory with all variables and numerals having the bit-width 32. The formula contains one unconstrained variable x. The term $x + (3 \times y)$ is unconstrained as it can attain any value, regardless of the value of y. Indeed, any bit-vector value $bv \in \{0,1\}^{32}$ can be obtained from $x + (3 \times y)$ by setting $x = -(3 \times y) + bv$. We can thus replace the term $x + (3 \times y)$ by a fresh variable v and get an equisatisfiable formula $(v = 0 \wedge y >_u 0)$. Alternatively, one can realize that the whole term $x + (3 \times y) = 0$ is unconstrained and thus it can be replaced by a fresh *Bool* variable w. In this way, we get an equisatisfiable formula $(w \wedge y >_u 0)$. In both cases, the variable y of the simplified formula becomes unconstrained and the formula can be simplified further.

Recall that formulas are terms of sort Bool.

To formalize the simplification principle, we define when a term is *unconstrained due to* a set of variables U, which means that a term can evaluate to an arbitrary value by changing only values of variables in U. Further, we define when a term is unconstrained in a formula φ, which means that it is unconstrained due to a set of variables that are unconstrained in φ.

Definition 7.1 (Unconstrained term). *Let t be a term and $U \subseteq vars(t)$ be a set of variables. We say that the term t is* unconstrained due to U *if for each valuation μ of variables in $(vars(t) \smallsetminus U)$ and every value b of the same sort as the term t, there exists a valuation ν of variables in U such that $[\![t]\!]_{\mu \cup \nu} = b$.*

Example 7.1. *In the bit-vector theory, the following terms are unconstrained due to $\{x\}$ for any term t' not containing x:*

- $x^{[n]} + t'^{[n]}$ *and* $t'^{[n]} + x^{[n]}$,
- $c^{[n]} \times x^{[n]}$ *and* $x^{[n]} \times c^{[n]}$ *if c is an odd numeral,*
- $\sim x^{[n]}$,
- $x^{[n]} <_u c^{[n]}$ *if* $c \neq 0$,
- $c^{[n]} <_u x^{[n]}$ *if* $c \neq 2^n - 1$,
- $x^{[n]} = t'^{[n]}$ *and* $x^{[n]} \neq t'^{[n]}$.

Note that the last two terms are unconstrained due to $\{x^{[n]}\}$ because each sort of the bit-vector theory contains at least two elements. Further, the terms $x^{[n]} \times y^{[n]}$, $(x^{[n]} \,\&\, y^{[n]})$, $(x^{[n]} \mid y^{[n]})$ are unconstrained due to $\{x^{[n]}, y^{[n]}\}$. A comprehensive list of unconstrained terms can be found for example in Franzén's doctoral thesis [Fra10].

On the contrary, multiplication by an even numeral is not an unconstrained term. For example, the term $2^{[32]} \times x^{[32]}$ over the theory of bit-vectors never evaluates to $bv_{32}(3)$ as the least significant bit of the results has to be 0. In other words, the number 3 does not have a multiplicative inverse in the ring of integers modulo 2^{32}. As a consequence, the term $x^{[32]} \cdot y^{[32]}$ is neither unconstrained due to $\{x^{[32]}\}$, nor unconstrained due to $\{y^{[32]}\}$.

As was stated in the introduction of this chapter, unconstrained terms are useful when they are unconstrained due to a set of variables that occur only once in the given formula. We call such terms unconstrained in the given formula.

Definition 7.2 (Unconstrained term)**.** *A subterm t of a formula φ is called* unconstrained in the formula φ *if it is unconstrained due to a set of variables that are unconstrained in the formula φ.*

The following theorem states the correctness of simplification based on unconstrained terms.

Theorem 7.1 ([Bru08; Fra10])**.** *Let φ be a quantifier-free formula and t its subterm unconstrained in φ. Then φ is equisatisfiable with the formula $\varphi[t \leftarrow v]$, where v is a fresh variable of the same sort as t.*

Note that our definition of unconstrained terms and the statement of Theorem 7.1 are slightly more general than the ones given by Brummayer and Bruttomesso, which consider unconstrained terms containing only a single unconstrained variable. The definition of an unconstrained term used in this chapter is due to Franzén [Fra10].

The definition of many-sorted logic, in which subformulas are identified with terms of sort *Bool*, brings some additional benefits. In particular, a subformula can be an unconstrained term even if it consists of terms that are not unconstrained. For example, let us consider the formula $\varphi \equiv (3x+3y = 0 \wedge y > 0)$ over the theory of integers. The term $3x + 3y$ is not unconstrained as its value is always a multiple of 3. However, term $3x + 3y = 0$ of sort *Bool* is unconstrained due to $\{x\}$ because it can evaluate to 1 by setting $x \mapsto -y$ and to 0 by setting $x \mapsto -y + 1$. As x is unconstrained in φ, we can simplify the formula to the equisatisfiable form $(v \wedge y > 0)$. Elimination of pure literals [DLL62] can then further reduce the formula to the form $(y > 0)$. As both 1 and 0 can be obtained by suitable choices of the value of the variable y, the term $y > 0$ is unconstrained due to $\{y\}$, and thus the formula can be simplified to v', where v' is a Boolean variable.

7.1.1 *Note on models*

The simplified formulas are in general equisatisfiable to the original ones, but not equivalent. For example, the formulas $(x + (3 \times y) = 0 \wedge y > 0)$ and $(v = 0 \wedge y > 0)$ mentioned above are both satisfiable, but they use different sets of variables and thus they have different models. In this case, a model of the original formula can be easily computed from the model μ of the simplified formula: it assigns to y the value $[\![y]\!]_\mu$ and to x the value $[\![-(3 \times y)]\!]_\mu$. However, in some cases, the computation of a model for the original formula can be harder. For example, assume that we have replaced the unconstrained term $180423^{[32]} \cdot x$ over the bit-vector theory by a fresh variable y. To get the value of x such that the term $180423^{[32]} \cdot x$ evaluates to a given value of y then means to find the multiplicative inverse of 180423 in the ring of integers modulo 2^{32} and multiply it by the value of y. Although this inverse can be still computed using an extended Euclidean algorithm [Bur97], it is computationally not trivial.

Note that algorithms for effective retrieval of models for the original formulas from models of the simplified formulas are beyond the scope of this thesis.

7.2 PARTIALLY CONSTRAINED TERMS

The key property of the simplification presented in the previous section is that the possible values of an unconstrained term are precisely the same as the possible values of a fresh variable of the same sort. This approach can be generalized even to terms that are *partially constrained*: a complex term can be replaced by a simpler one representing the same values. For example, the value of the term $6^{[n]} \times x^{[n]}$ over the bit-vector theory can be any bit-vector divisible by $2^{[n]}$. Therefore, if $6^{[n]} \times x^{[n]}$ is a subterm of a formula where $x^{[n]}$ is unconstrained, then the subterm $6^{[n]} \times x^{[n]}$ can be replaced by $2^{[n]} \times y^{[n]}$ where $y^{[n]}$ is a fresh variable of the same bit-width as $x^{[n]}$.

The following definition formalizes the notion that two terms represent the same set of possible values for any fixed valuation of variables in C. Intuitively, in applications of this definition, the set C will contain all variables in the input formula that are not unconstrained, because their values could be enforced by the rest of the formula. Such variables, which are not unconstrained in the given formula, are called *constrained* in this formula.

Definition 7.3 (C-interchangeable terms). *Let C be a set of variables and t, s be terms of the same sort. Further, let $U = (vars(t) \cup vars(s)) \setminus C$. Terms t and s are called* C-interchangeable, *written $t \overset{C}{\rightleftharpoons} s$, if for every valuation μ of variables in C it holds that*

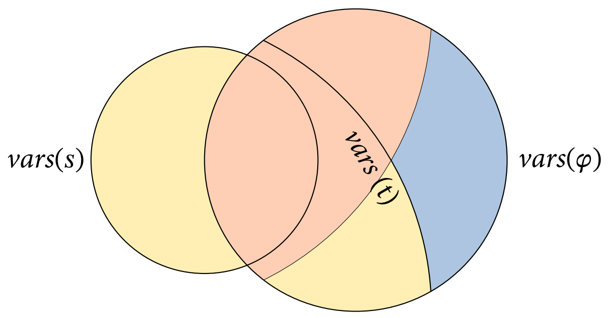
$$\{[\![t]\!]_{\mu \cup \nu} \mid \nu \text{ is a valuation of } U\} = \{[\![s]\!]_{\mu \cup \rho} \mid \rho \text{ is a valuation of } U\}.$$

Now we precisely formulate the simplification principle for partially constrained terms and prove its correctness.

Theorem 7.2. *Let φ be a quantifier-free formula, t its subterm, and C a superset of all constrained variables in φ that occur in t. Then for any term s such that $t \overset{C}{\rightleftharpoons} s$ and $vars(s) \cap vars(\varphi) \subseteq C$, the formula φ is equisatisfiable with the formula $\varphi[t \leftarrow s]$.*

Proof. As is illustrated by the following diagram, all variables of φ and s can be divided into three disjoint sets:

1. the set C that contains all constrained variables in t (marked red in the diagram),
2. the set $U = (vars(t) \cup vars(s)) \setminus C$ of all variables in t or s that are not in C, i.e., the set of all variables in s that are not in φ and some unconstrained variables of t (marked yellow in the diagram),
3. the set U' containing all variables that are neither in C nor in U, i.e., a set unconstrained variables of the original formula that do not occur in t (marked blue in the diagram).



The precondition $vars(s) \cap vars(\varphi) \subseteq C$ formulated in the theorem implies that every variable of U appears either only in s or only in t and not in any other part of the formula. Moreover, variables of U' appear neither in term t, nor in term s.

Suppose that φ is satisfiable. Hence, there exists a valuation μ of variables in C, a valuation ν of variables in U, and a valuation ν' of variables in U' such that $\mu \cup \nu \cup \nu'$ is a satisfying assignment of φ. Because t does not contain any variable from U', we know that $[\![t]\!]_{\mu\cup\nu\cup\nu'} = [\![t]\!]_{\mu\cup\nu}$. As t and s are C-interchangeable, there exists a valuation ρ of variables in U such that $[\![t]\!]_{\mu\cup\nu} = [\![s]\!]_{\mu\cup\rho}$. As valuations μ and ρ concern only variables that do not appear in U', the result of $\mu \cup \rho \cup \nu'$ is a well-defined function that assigns values to all variables in $C \cup U \cup U'$. Finally, because $[\![s]\!]_{\mu\cup\rho} = [\![s]\!]_{\mu\cup\rho\cup\nu'}$ and because variables in U do not occur in the formula $\varphi[t \leftarrow s]$, we get that the assignment $\mu \cup \rho \cup \nu'$ satisfies $\varphi[t \leftarrow s]$.

It remains to show that satisfiability of $\varphi[t \leftarrow s]$ implies satisfiability of φ. However, the arguments are completely symmetric. $\square$

Note that the definition of C-interchangeability generalizes the previous Definition 7.1 in the sense that a term t is unconstrained due to U if and only if it is C-interchangeable with a fresh variable u of the same sort as t for the set $C = vars(t) \setminus U$. Theorem 7.1 can then be seen as a corollary of Theorem 7.2.

7.2.1 *Applications*

Now we show some applications of the previous theorem. We start with an example from the theory of non-linear real arithmetic and then focus on terms over the bit-vector theory. In particular, we focus on simplification of partially constrained terms with multiplication as this operation is very expensive for some SMT solvers, especially these based on BDDs.

Example 7.2. *Consider the term $t \times u$ in the theory of non-linear real arithmetic, where u is an unconstrained variable and t is an arbitrary term not containing the variable u. The term $t \times u$ can be replaced by $\texttt{ite}(t = 0, 0, v)$, where v is a fresh variable, as the terms $t \times u$ and $\texttt{ite}(t = 0, 0, v)$ are $vars(t)$-interchangeable. Indeed, if $[\![t \times u]\!]_{\mu\cup\{u\mapsto n_u\}} = n$, then the same value n can be obtained from the term $\texttt{ite}(t = 0, 0, v)$ by setting v to $[\![t]\!]_\mu \times n_u$. On the other hand, if $[\![\texttt{ite}(t = 0, 0, v)]\!]_{\mu\cup\{v\mapsto n_v\}} = n'$, then the same value n' can be obtained from the term $t \times u$ by setting u to 0 if $[\![t]\!]_\mu$ is 0 and by setting it to $\frac{n_v}{[\![t]\!]_\mu}$ otherwise.*

In general, this simplification can be performed in any theory in which addition and multiplication form a field.

Example 7.3. *In the bit-vector theory, the term $4 \times u$ can be evaluated to any bit-vector where the two least significant bits are zeroes. The same holds for the term $12 \times u$. In general, the term $c^{[n]} \times u$ with a numeral $c^{[n]}$ can represent any bit-vector ending with i zeroes, where i is the greatest integer such that 2^i divides c. This follows from the fact that c can be expressed as $2^i \times m$ for some odd number m and every odd number has a multiplicative inverse m^{-1} in the bit-vector theory. All bit-vectors with i zeroes at the end can be also represented by the term $v \ll i$. Hence, the terms $c^{[n]} \times u$ and $v \ll i$ are $\emptyset$-interchangeable. Finally, Theorem 7.2 implies that a formula φ with an unconstrained variable u and a term $c^{[n]} \times u$ is equisatisfiable with the formula $\varphi[(c^{[n]} \times u) \leftarrow (v^{[n]} \ll i)]$ where $v^{[n]}$ is a fresh variable and i is the constant described above. Note that the term $v \ll i$ is easier to compute and express as a circuit than the original multiplication by a potentially large and complicated number c.*

$\emptyset$-interchangeability means that the two interchanged terms do not share any variables.

Example 7.4. *More interestingly, we can simplify also the term $t \times u$ where u is unconstrained, even if t not a numeral. As an example, consider the term $t \times u^{[3]}$ for a 3-bit variable u. We write $t[i]$ as a shortcut for the extraction of the i-th least significant bit of the term t where $0 \leq i \leq 2$, i.e., $t[i] \equiv \mathtt{extract}_{i,i}(t)$. Then $t \times u^{[3]}$ is vars(t)-interchangeable with the term*

$$\begin{aligned}\mathtt{ite}\Big(t[0] = 1^{[1]},\quad & v,\\ & \mathtt{ite}\big(t[1] = 1^{[1]},\quad v \ll 1^{[3]},\\ & \qquad \mathtt{ite}(t[2] = 1^{[1]},\ v \ll 2^{[3]},\ 0^{[3]})\big)\Big).\end{aligned}$$

This term uses the idea from the previous example, but performs dynamic choice on the number of the least significant zeroes in the value of t. If the last significant bit of t is 1, then any value can be obtained from $t \times u^{[3]}$ by choosing a suitable value of $u^{[3]}$. Additionally, if the least significant bit of t is 0 and the second least significant bit is 1, then any value with the least significant bit 0 can be obtained from $t \times u^{[3]}$. Similarly, any value with the two least significant bits can be obtained from $t \times u^{[3]}$ if the three least significant bits of t are 100. Finally, if the three least significant bits of t are 000 then the result of $t \times u^{[3]}$ must be $0^{[3]}$.

In general, the term $t \times u^{[n]}$ is vars(t)-interchangeable with the term defined as

$$\begin{aligned}\mathtt{ite}\Big(t[0] = 1^{[1]},\quad & v,\\ & \mathtt{ite}\big(t[1] = 1^{[1]},\quad v \ll 1^{[n]},\\ & \qquad \dots\\ & \qquad \mathtt{ite}(t[k-1] = 1^{[1]},\quad v \ll (k-1)^{[n]},\\ & \qquad\qquad 0^{[n]})\dots\big)\Big).\end{aligned}$$

Therefore, in a formula φ with an unconstrained variable $u^{[n]}$, a term $t \times u^{[n]}$ can be replaced by the term above with a fresh variable v and the resulting formula is equisatisfiable with φ.

$t <_u u^{[n]}$	$b \wedge t \neq \mathit{unsignedMax}^{[n]}$
$t <_s u^{[n]}$	$b \wedge t \neq \mathit{signedMax}^{[n]}$
$u^{[n]} <_u t$	$b \wedge t \neq 0^{[n]}$
$u^{[n]} <_s t$	$b \wedge t \neq \mathit{signedMin}^{[n]}$
$t \leq_u u^{[n]}$	$b \vee t = 0^{[n]}$
$t \leq_s u^{[n]}$	$b \vee t = \mathit{signedMin}^{[n]}$
$u^{[n]} \leq_u t$	$b \vee t = \mathit{unsignedMax}^{[n]}$
$u^{[n]} \leq_s t$	$b \vee t = \mathit{signedMax}^{[n]}$

Table 7.1: Each line presents a pair of $\mathit{vars}(t)$-interchangeable terms, assuming that $b, u \notin \mathit{vars}(t)$. Terms on the right are considered simpler for SMT solvers than these on the left.

This term was not mentioned in our original paper [JS17] and it is inspired by the corresponding invertibility condition for multiplication [Nie+18b].

Using known bitwise tricks[1]*, a much simpler $\mathit{vars}(t)$-interchangeable term can be defined. The term $t \times u^{[n]}$ is $\mathit{vars}(t)$-interchangeable with the term $v \,\&\, (t \mid -t)$. Recall that the result of $(t \mid -t)$ has the same number of least significant zeroes as the value of t and has all the other bits set to 1.*

In the previous two examples, the multiplications have been replaced by bitwise operations and bit-shift by a constant, which are generally cheaper for the SMT solvers.

Example 7.5. *Now we discuss several simplification rules mentioned by Bruttomesso without a proof of correctness. For example, consider the simplification rule that rewrites the term $t >_u u^{[n]}$ containing an unconstrained bit-vector variable u to the term $b \wedge t \neq 0^{[n]}$, where b is a fresh Bool variable. Intuitively, the rule is correct as $t >_u u^{[n]}$ can be evaluated to both 1 and 0 unless t is evaluated to 0. If the value of t is 0, $t >_u u^{[n]}$ evaluates to 0. Correctness of this rule follows directly from Theorem 7.2 and the fact that the term $t >_u u^{[n]}$ is $\mathit{vars}(t)$-interchangeable with the term $b \wedge t \neq 0^{[n]}$ assuming that $u^{[n]}, b \notin \mathit{vars}(t)$. Similar simplification rules can be derived from pairs of $\mathit{vars}(t)$-interchangeable terms presented in Table 7.1.*

7.3 UNCONSTRAINED TERMS IN QUANTIFIED FORMULAS

In this section, we extend the treatment of unconstrained variables to formulas containing quantifiers. To simplify the presentation, we suppose that all formulas are in the prenex normal form and do not contain any free variables. That is, $\varphi = Q_1 B_1 Q_2 B_2 \ldots Q_n B_n \psi$, where ψ is a quantifier-free formula, $Q_i \in \{\forall, \exists\}$ for all $1 \leq i \leq n$, and all B_i are pairwise disjoint sets of variables. Sequences $Q_i B_i$ are called *quantifier blocks*. Quantifier blocks are supposed to be maximal, that is $Q_i \neq Q_{i+1}$. A quantifier block $Q_i B_i$ is *existential* if $Q_i = \exists$

1 https://catonmat.net/low-level-bit-hacks

and *universal* otherwise. The *level of a variable* x is i such that $x \in B_i$. For a variable x, we denote as $level(x)$ its level and for a set of variables X we define $levels(X) = \{level(x) \mid x \in X\}$. If the set X contains only variables of the same level, we denote as $level(X)$ the level of all variables in that set. A variable is called *unconstrained* in the quantified formula φ if it is unconstrained in its quantifier-free part ψ.

It is easy to see that Theorem 7.1 can not be directly applied to quantified formulas. As an example, consider the formula $\varphi \equiv \exists x \forall y\,(x + y = 0)$ where both variables have bit-width 32. Although the variable x is unconstrained in the formula φ and the term $x+y$ is unconstrained due to $\{x\}$, the conclusion of Theorem 7.1 is not true regardless of the position of the quantifier for the fresh variable v: the formula φ is equisatisfiable neither with $\exists v \forall y\,(v = 0)$ nor with $\forall y \exists v\,(v = 0)$. Intuitively, the problem is with the order of the quantifiers: the simplification of unconstrained terms relies on the ability to choose a suitable value of the unconstrained variable x for each value of y. However, in the mentioned example, the variable y is quantified *after* the variable x and therefore its value is unknown in the time when we have to choose the suitable value of the variable x.

The following modified definition of the unconstrained term solves this problem by forbidding any unconstrained variable in the given term to be quantified before any constrained variable of the term.

Definition 7.4 (Quantified unconstrained term)**.** *Let φ be a quantified formula, t its subterm, and $U \subseteq vars(t)$ a set of variables such that $|levels(U)| = 1$. We say that the term t is* unconstrained due to U *if for each valuation μ of variables in $(vars(t) \setminus U)$ and every value b of the same sort as the term t, there exists a valuation ν of variables in U such that $[\![t]\!]_{\mu\cup\nu} = b$ and, furthermore,*

$$level(U) \geq \max\Big(levels\big(vars(t) \setminus U\big)\Big).$$

For example, in the formula $\exists x \forall y\,(x+y = 0)$ mentioned above, the subterm $x + y$ is not unconstrained due to $\{x\}$, since $level(x) < level(y)$. On the other hand, it is unconstrained due to $\{y\}$.

The following theorem shows that subterms that are unconstrained due to a set of unconstrained variables can be simplified even in quantified formulas.

Theorem 7.3. *Let φ be a formula, t be a term, U be a subset of $vars(t)$, and v be a variable not occurring in φ. If t is unconstrained due to the set of variables U and all variables in U are unconstrained in φ, then φ is equivalent to the formula φ in which the term t is replaced by v and the variables in U are replaced in their quantifier block by the variable v.*

Proof. The definition of unconstrained subterm implies that all variables in U have the same level. Let $k = level(U)$ and let $\varphi \equiv Q_1 B_1 \ldots Q_k B_k\, \psi$, where the formula ψ can contain quantifiers. We show that the formula $Q_k B_k\, \psi$ is equivalent to the formula $Q_k((B_k \setminus U) \cup \{v\})\,(\psi[t \leftarrow v])$ and thus also the formula φ is equivalent to $Q_1 B_1 \ldots Q_{k-1} B_{k-1} Q_k((B_k \setminus U) \cup \{v\})\,(\psi[t \leftarrow v])$ as required.

Let $V = \bigcup_{1 \leq i < k} B_i$. Observe that $U \subseteq B_k$ and the last line of the definition of an unconstrained term implies $(vars(t) \setminus U) \subseteq V \cup (B_k \setminus U)$. Let μ be an assignment of values to all variables in V. We distinguish two cases according to the quantifier Q_k.

- Suppose that $Q_k = \exists$. If $[\![\exists B_k \psi]\!]_\mu = 1$, then there is a valuation ν of variables in B_k such that $[\![\psi]\!]_{\mu\cup\nu} = 1$. Note that the function $\mu \cup \nu$ assigns values to a superset of $vars(t)$ and therefore can be used to evaluate the term t. Let b_v be the value $[\![t]\!]_{\mu\cup\nu}$. For this value, we have $[\![\psi[t \leftarrow v]]\!]_{\mu\cup\nu\cup\{v\mapsto b_v\}} = 1$ and therefore also $[\![\exists(B_k \cup \{v\})\psi[t \leftarrow v]]\!]_\mu = 1$. Since all variables in U are unconstrained, the formula $\psi[t \leftarrow v]$ does not contain any variable from U and therefore $[\![\exists((B_k \setminus U) \cup \{v\})\, \psi[t \leftarrow v]]\!]_\mu = 1$.

  Conversely, if $[\![\exists((B_k \setminus U) \cup \{v\})\, \psi[t \leftarrow v]]\!]_\mu = 1$, there is a valuation ν of variables in $(B_k \setminus U) \cup \{v\}$ such that $[\![\psi[t \leftarrow v]]\!]_{\mu\cup\nu} = 1$. As t is unconstrained due to the set U, there is a valuation ν_U of variables in U such that $[\![t]\!]_{\mu\cup\nu\cup\nu_U} = \nu(v)$. Therefore $[\![\psi]\!]_{\mu\cup\nu\cup\nu_U} = 1$ and in turn $[\![\exists(B_k \cup \{v\})\, \psi]\!]_\mu = 1$, because $\nu \cup \nu_U$ is an assignment to variables from $B_k \cup \{v\}$. Finally, because the formula ψ does not contain the variable v, we know that $[\![\exists B_k\, \psi]\!]_\mu = 1$.

- If $Q_k = \forall$, the proof is dual to the $\exists$ case, but each existential quantifier is replaced by the universal quantifier and each 1 is replaced by 0. □

As an example, consider again the formula $\exists x \forall y\, (x + y = 0)$. According to the previous theorem, it is equivalent to $\exists x \forall v\, (v = 0)$ because the term $x + y$ is unconstrained due to $\{y\}$. Moreover, as the term $v = 0$ is unconstrained due to $\{v\}$, the formula is equisatisfiable with $\exists x \forall p\, (p)$, where p is a *Bool* variable. This formula is trivially equivalent to $\bot$.

7.4 PARTIALLY CONSTRAINED TERMS IN QUANTIFIED FORMULAS

Both described extensions of unconstrained terms, i.e., partially constrained terms and unconstrained terms in quantified formulas, can be combined together in a fairly obvious way. The next theorem precisely describes this combination. The proof of this theorem is a straightforward combination of already presented proofs.

Theorem 7.4. *Let φ be a closed quantified formula in PNF and t be its subterm such that a subset U of unconstrained variables appearing in t satisfies $|levels(U)| = 1$ and*

$$level(U) \geq \max\big(levels(C)\big),$$

where $C = vars(t) \setminus U$. Further, let s be an arbitrary term such that $t \overset{C}{\rightleftharpoons} s$ and $vars(s) \cap vars(\varphi) \subseteq C$. Then the formula φ is equivalent with the formula φ where the term t is replaced by the term s and the variables of U are replaced in their quantifier block by the set of variables $vars(s) \setminus vars(\varphi)$.

Note that due to this theorem, we can easily transfer simplification rules mentioned by Bruttomesso to quantified formulas, because they can be reformulated using the notion of interchangeable terms, as was described in Section 7.2.

	Quantifier type of u	
Term	Existential	Universal
$u^{[n]} = t$ or $t = u^{[n]}$	$\top$	$\bot$
$u^{[n]} \neq t$ or $t \neq u^{[n]}$	$\top$	$\bot$
$t <_u u^{[n]}$	$t \neq \mathit{unsignedMax}^{[n]}$	$\bot$
$t <_s u^{[n]}$	$t \neq \mathit{signedMax}^{[n]}$	$\bot$
$u^{[n]} <_u t$	$t \neq 0^{[n]}$	$\bot$
$u^{[n]} <_s t$	$t \neq \mathit{signedMin}^{[n]}$	$\bot$
$t \leq_u u^{[n]}$	$\top$	$t = 0^{[n]}$
$t \leq_s u^{[n]}$	$\top$	$t = \mathit{signedMin}^{[n]}$
$u^{[n]} \leq_u t$	$\top$	$t = \mathit{unsignedMax}^{[n]}$
$u^{[n]} \leq_s t$	$\top$	$t = \mathit{signedMax}^{[n]}$

Table 7.2: Derived simplification rules for partially constrained terms (in the left column) with positive polarity in quantified formulas. We assume that u is an unconstrained variable and $\mathit{level}(u) \geq \max(\mathit{levels}(\mathit{vars}(t)))$.

Moreover, such simplifications can be combined with additional quantifier-specific simplification rules such as the elimination of pure literals. The key observation is that the simplifications using unconstrained and partially constrained terms often introduce fresh – and therefore unconstrained – *Bool* variables (see Table 7.1). Furthermore, if b is an existentially quantified *Bool* variable that is unconstrained in a formula φ, it can be replaced by $\top$ if it occurs in φ with the positive polarity and by $\bot$ if it occurs with the negative polarity and the resulting formula will be equivalent to the original one. Similarly, an unconstrained universally quantified *Bool* variable can be replaced by $\bot$ if it has the positive polarity and by $\top$ if it has the negative polarity. Combining those simplifications with simplifications using unconstrained and partially constrained terms therefore yields more straightforward simplification rules, which are shown in Table 7.2. Although this table shows only rules for terms with positive polarity, the dual versions for terms with negative polarity are straightforward.

7.5 GOAL UNCONSTRAINED TERMS

In the previous sections, we have seen that in some cases, a subterm t of a formula can be replaced by a simpler subterm that generates precisely the same set of results. We now improve this technique further by considering the context of the term t in the input formula. For example, consider the formula

$$\varphi \equiv \varphi' \wedge (u \times t_1 \leq_u t_2),$$

where t_1 and t_2 are arbitrary terms of bit-width 32, φ' is an arbitrary quantifier-free formula, and the variable u is unconstrained. According to Example 7.4, the subterm $u \times t_1$ can be replaced by the term $v \mathbin{\&} (t_1 \mid -t_1)$. However, the subterm $u \times t_1$ can actually be replaced by an even simpler term because it occurs as the first argument of the predicate $\leq_u$. The term $u \times t_1$ hence can be replaced by a term that describes the least possible unsigned value that can be result of $u \times t_1$ for any choice of the value u. Namely, the subterm $u \times t_1$ can be replaced by the subterm $0^{[32]}$, because it can be obtained by setting u to $bv_{32}(0)$. The intuition for the correctness of this rewrite is as follows: if the entire formula should be satisfiable, the subterm $u \times t_1 \leq_u t_2$ will certainly be satisfied by the least possible value of $u \times t_1$; if, on the other hand, the least possible value of $u \times t_1$ satisfies the formula, it is certainly satisfiable. Note that the value of the variable u can be changed arbitrarily, e.g., set to $bv_{32}(0)$, without affecting the rest of the formula, since the variable u is unconstrained.

This section formalizes the above-mentioned intuition of replacing a term by simpler term that captures its least or greatest possible value. We call terms that can be replaced in this way as *goal unconstrained terms*. Similarly to the previous sections, which introduced partially unconstrained terms, we first formalize goal unconstrained terms in quantifier-free formulas and extend the treatment to quantified formulas later-on.

We begin by formalizing the subterms of the input formula whose unsigned value can be increased or decreased while preserving satisfiability of the input formula; we call such subterms *unsigned-monotone* and *unsigned-antimonotone*, respectively.

Definition 7.5 (Unsigned-monotone and unsigned-antimonotone occurrence)**.** *Let φ be an arbitrary quantifier-free formula and t be its subterm. We say that the subterm t* occurs unsigned-monotonically *in φ if for arbitrary terms s and v of the same sort as t and each valuation μ of variables in t, s, and v, the following holds:*

$$\text{if } [\![s \leq_u v]\!]_\mu = 1 \text{ and } \mu \text{ satisfies } \varphi[t \leftarrow s], \text{ then } \mu \text{ satisfies } \varphi[t \leftarrow v].$$

We say it occurs unsigned-antimonotonically *in φ, if all such terms t, s, and v and each valuation μ satisfy:*

$$\text{if } [\![v \leq_u s]\!]_\mu = 1 \text{ and } \mu \text{ satisfies } \varphi[t \leftarrow s], \text{ then } \mu \text{ satisfies } \varphi[t \leftarrow v].$$

As mentioned in the motivation, one group of unsigned-monotone subterms t are the ones that occur only in the subformulas of form $s \leq_u t$ that have the positive polarity. For each such term t, its unsigned value can be arbitrarily increased without turning the satisfying assignment to an unsatisfying one. Similarly, terms t that occur only in subformulas that have the positive polarity and are of the form $t \leq_u s$ are unsigned-antimonotone. Dually, in subformulas with the negative polarity, each term t that occurs only in $s \leq_u t$ is unsigned-antimonotone and the term t in $t \leq_u s$ is unsigned-monotone. Furthermore, the unsigned-monotonicity and unsigned-antimonoticity can be propagated through the monotonic operations that preserve the ordering.

For example, if $\mathtt{concat}(t, s)$ occurs unsigned-monotonically in φ, then the subterms t and s also occur unsigned-monotonically in φ, since the formulas $x \leq_u x' \rightarrow \mathtt{concat}(x, y) \leq_u \mathtt{concat}(x', y)$ and $y \leq_u y' \rightarrow \mathtt{concat}(x, y) \leq_u \mathtt{concat}(x, y')$ are valid.

We now define the necessary notions of *unsigned-min* and *unsigned-max C-interchangeability*, which capture the mentioned property that two terms have exactly the same possible least unsigned result and greatest unsigned result, respectively, for all values of variables in C. For this, we use the function $\min_u$, which to each set of bit-vectors of the same bit-width returns the bit-vector in this set with the least unsigned value, and the function $\max_u$, which similarly for a set returns its bit-vector with the greatest unsigned value.

Definition 7.6 (Unsigned-min and unsigned-max C-interchangeability)**.** *Let C be a set of variables and t, s be terms of the same sort. Further, let $U = (vars(t) \cup vars(s)) \setminus C$. Terms t and s are called* unsigned-min C-interchangeable *if for every valuation μ of variables in C it holds that*

$$\min_u(\{[\![t]\!]_{\mu \cup \nu} \mid \nu \textit{ is a valuation of } U\}) = \min_u(\{[\![s]\!]_{\mu \cup \rho} \mid \rho \textit{ is a valuation of } U\}).$$

Similarly, the terms t and s are called unsigned-max C-interchangeable *if for every valuation μ of variables in C it holds that*

$$\max_u(\{[\![t]\!]_{\mu \cup \nu} \mid \nu \textit{ is a valuation of } U\}) = \max_u(\{[\![s]\!]_{\mu \cup \rho} \mid \rho \textit{ is a valuation of } U\}).$$

Example 7.6. *As the motivating example showed, if $u \notin vars(t)$, the terms $u^{[n]} \times t$ and $0^{[n]}$ are unsigned-min $vars(t)$-interchangeable, because for each valuation μ of variables in $vars(t)$, the equation*

$$\min_u\left(\{[\![u^{[n]} \times t]\!]_{\mu \cup \nu} \mid \nu \textit{ is a valuation of } \{u^{[n]}\}\}\right) = bv_n(0)$$

can be shown to hold by considering $\nu = \{u^{[n]} \mapsto bv_n(0)\}$.

We now show that the proposed simplification method, which allows replacing unsigned-min and unsigned-max C-interchangeable terms that contain unconstrained variables is indeed correct.

Theorem 7.5. *Let φ be a quantifier-free formula, t its subterm that occurs unsigned-antimonotonically in φ, and C be a superset of all constrained variables that occur in t. Then for any term s that is unsigned-min C-interchangeable with t and that satisfies $vars(s) \cap vars(\varphi) \subseteq C$, the formula φ is equisatisfiable with the formula $\varphi[t \leftarrow s]$.*

Proof. The structure of the proof is similar to the one of Theorem 7.2. Again, all variables of φ and s can be divided into three disjoint sets:

1. the set C that contains all constrained variables in t,
2. the set $U = (vars(t) \cup vars(s)) \setminus C$ of all variables in t or s that are not in C,
3. the set U' containing all variables that are neither in C nor in U.

Recall that the set U contains only variables that occur only in t or only in s.

Let n be the bit-width of the terms t and s. Suppose that φ is satisfiable. Hence, there exist valuations φ, ν, and ν' of variables in C, U, and U', respectively, such that $\mu \cup \nu \cup \nu'$ satisfies φ. Because t does not contain any variable from U', we know that $[\![t]\!]_{\mu\cup\nu\cup\nu'} = [\![t]\!]_{\mu\cup\nu}$. Because ν is a valuation of U, we know that

$$nat_n(\min_u(\{[\![t]\!]_{\mu\cup\nu} \mid \nu \text{ is a valuation of } U\})) \;\leq\; nat_n([\![t]\!]_{\mu\cup\nu}).$$

And because t and s are unsigned-min C-interchangeable, there is an assignment ρ to the variables of U such that $nat_n([\![s]\!]_{\mu\cup\rho}) \leq nat_n([\![t]\!]_{\mu\cup\nu})$. We can suppose that the assignment ρ agrees with the assignment ν on all variables from $vars(t) \setminus C$, because they do not occur in the term s. Therefore, we know that $[\![s \leq_u t]\!]_{\mu\cup\rho\cup\nu'} = 1$ and that the assignment $\mu \cup \rho \cup \nu'$ satisfies φ. And because it satisfies $\varphi \equiv \varphi[t \leftarrow t]$, the term t occurs unsigned-antimonotonically in φ, and $[\![s \leq_u t]\!]_{\mu\cup\rho\cup\nu'} = 1$, the formula $\varphi[t \leftarrow s]$ must also satisfied by $\mu \cup \rho \cup \nu'$.

The arguments to show that satisfiability of $\varphi[t \leftarrow s]$ implies the satisfiability of φ are again completely symmetric. □

Observe that the dual theorem allows replacing each term that occurs unsigned-monotonically in a formula by a term with which it is unsigned-max C-interchangeable. Moreover, all the presented notions have the straightforward corresponding counterparts for the comparison of *signed* integers. We thus obtain notions of *signed-monotone occurence*, *signed-antimonotone occurence*, *signed-min C-interchangeability*, and *signed-max C-interchangeability*, which satisfy a claim analogous to the one of the Theorem 7.5.

7.5.1 *Applications*

We now present more examples of bit-vector unsigned-min and unsigned-max C-interchangeable terms. We focus on operations like division and remainder, which are hard for most of the current SMT solvers. We also focus on bit-shifts, because they are hard for BDD based SMT solvers as they introduce multiple dependencies between bits on different positions in the input bit-vectors.

Example 7.7. *Consider the term $t /_u u^{[n]}$ for an arbitrary term t and an arbitrary assignment μ to the variables in t.*

The least possible unsigned result of $t /_u u^{[n]}$ can be obtained by setting $u^{[n]}$ to $unsignedMax^{[n]}$. If the value of $[\![t]\!]_\mu$ is equal to $unsignedMax^{[n]}$, the least possible unsigned result of $t /_u u^{[n]}$ is thus $bv_n(1)$ and otherwise it is $bv_n(0)$. The term $t /_u u^{[n]}$ is thus unsigned-min $vars(t)$-interchangeable with the term `ite`$(t = unsignedMax^{[n]}, 1^{[n]}, 0^{[n]})$.

Recall that $[\![t /_u 0^{[n]}]\!]_\mu = unsignedMax^{[n]}$.

On the other hand, the greatest unsigned value obtainable from $t /_u u^{[n]}$ is $unsignedMax^{[n]}$, which is obtained by setting $u^{[n]}$ to $bv_n(0)$. The term $t /_u u^{[n]}$ is thus unsigned-max $vars(t)$-interchangeable with the term $unsignedMax^{[n]}$.

Example 7.8. *Consider the term $u^{[n]} \%_u t$ for an arbitrary term t and an arbitrary assignment μ to the variables in t.*

The least possible unsigned value of $u^{[n]} \%_u t$ can be obtained by setting $u^{[n]}$ to $bv_n(0)$. This works even for the case when the value of t is $bv_n(0)$ because $[\![t \%_u 0^{[n]}]\!]_\mu = [\![t]\!]_\mu$. The term $u^{[n]} \%_u t$ is thus unsigned-min $vars(t)$-interchangeable with the term $0^{[n]}$.

On the other hand, the greatest possible unsigned value of the term $u^{[n]} \%_u t$ is $[\![t - 1^{[n]}]\!]_\mu$. To see this, consider two cases. If $[\![t]\!]_\mu$ is $bv_n(0)$, then the result of $u^{[n]} \%_u t$ is its first argument, which may be $unsignedMax^{[n]}$. However, $unsignedMax^{[n]}$ is exactly $[\![0^{[n]} - 1^{[n]}]\!]_\mu$ due to underflows. If the value $[\![t]\!]_\mu$ is not zero, then the result of $u^{[n]} \%_u t$ is equal to the standard remainder of the natural numbers corresponding to the value of $u^{[n]}$ and the result of $[\![t]\!]_\mu$. This result is definitely at most $[\![t - 1^{[n]}]\!]_\mu$ and this value can be indeed obtained by setting $u^{[n]}$ to $[\![t - 1^{[n]}]\!]_\mu$. Putting this together, the term $u^{[n]} \%_u t$ is unsigned-max $vars(t)$-interchangeable with the term $t - 1^{[n]}$.

Example 7.9. *Consider the term $t \gg_u u^{[n]}$ for an arbitrary term t and an arbitrary assignment μ to the variables in t.*

The least possible unsigned value of $t \gg_u u^{[n]}$ is $bv_n(0)$ because it can be obtained by setting $u^{[n]}$ to $bv_n(n)$. The term $t \gg_u u^{[n]}$ is thus unsigned-min $vars(t)$-interchangeable with the term $0^{[n]}$.

On the other hand, shifting a bit-vector logically to the right by one or more bits cannot increase its unsigned value. The greatest possible unsigned value of the term $t \gg_u u^{[n]}$ is therefore $[\![t]\!]_\mu$, which can be obtained by setting $u^{[n]}$ to $bv_n(0)$. The term $t \gg_u u^{[n]}$ is thus unsigned-max $vars(t)$-interchangeable with the term t.

More unsigned-min and unsigned-max C-interchangeable terms are presented in Table 7.3. Note that each of these examples gives rise to a corresponding simplification rule.

7.6 GOAL UNCONSTRAINED TERMS IN QUANTIFIED FORMULAS

This section briefly shows how to extend the simplifications of goal unconstrained terms to quantified formulas. Similarly to the partially constrained terms in quantified formulas, the simplification depends on the quantifier type of the unconstrained variables of the term that is being rewritten. The treatment of terms whose unconstrained variables are bound existentially is given by Theorem 7.5. On the other hand, for the treatment of terms whose unconstrained variables are bound universally, we need the following theorem, which is the exact dual of Theorem 7.5 and whose proof is also analogous. Note that the definitions of unsigned-monotonicity and unsigned-antimonotonicity are dual: a term t occurs unsigned-monotonically in a formula φ if and only if it occurs unsigned-antimonotically in the formula $\neg\varphi$.

Theorem 7.6. *Let φ be a quantifier-free formula, t its subterm that occurs unsigned-monotically in φ, and C be a superset of all constrained variables that*

	$vars(t)$-interchangability	
Term	Unsigned-min	Unsigned-max
$u^{[n]} \times t$	$0^{[n]}$	$(t \mid -t)$
$t /_u u^{[n]}$	$\texttt{ite}(t = \mathit{unsignedMax}^{[n]}, 1^{[n]}, 0^{[n]})$	$\mathit{unsignedMax}^{[n]}$
$u^{[n]} /_u t$	$\texttt{ite}(t = 0^{[n]}, \mathit{unsignedMax}^{[n]}, 0^{[n]})$	$\mathit{unsignedMax}^{[n]} /_u t$
$t \%_u u^{[n]}$	$0^{[n]}$	t
$u^{[n]} \%_u t$	$0^{[n]}$	$t - 1^{[n]}$
$t \ll u^{[n]}$	$0^{[n]}$	?
$u^{[n]} \ll t$	$0^{[n]}$	$\mathit{unsignedMax}^{[n]} \ll t$
$t \gg_u u^{[n]}$	$0^{[n]}$	t
$u^{[n]} \gg_u t$	$0^{[n]}$	$\mathit{unsignedMax}^{[n]} \gg_u t$
$t \gg_s u^{[n]}$	$\texttt{ite}(t[n-1] = 0^{[1]}, 0^{[n]}, t)$	$\texttt{ite}(t[n-1] = 0^{[1]}, t, \mathit{unsignedMax}^{[n]})$
$u^{[n]} \gg_s t$	$0^{[n]}$	$\mathit{unsignedMax}^{[n]}$

Table 7.3: The table shows *unsigned-min* and *unsigned-max* $vars(t)$-interchangeable terms for selected bit-vector operations. In all the cases t is an arbitrary term of bit-width n. The *question mark* denotes that the corresponding $vars(t)$-interchangeable term is too complex to produce a reasonable simplification rule.

occur in t. Then for any term s that is unsigned-min C-interchangeable with t and that satisfies $vars(s) \cap vars(\varphi) \subseteq C$, the formula φ is valid if and only if the formula $\varphi[t \leftarrow s]$ is valid.

We now show an example of application of this theorem on a formula that is universally quantified.

Example 7.10. *Consider a formula $\varphi \equiv \forall x^{[n]} \forall y^{[n]} \forall u^{[n]} ((x^{[n]} \times y^{[n]}) \leq_u (u^{[n]} \times y^{[n]}))$, where the variable $u^{[n]}$ is unconstrained. Because the term $u^{[n]} \times y^{[n]}$ occurs unsigned-monotonically in $x^{[n]} \times y^{[n]} \leq_u u^{[n]} \times y^{[n]}$ and because $u^{[n]} \times y^{[n]}$ is unsigned-min $\{y^{[n]}\}$-interchangeable with $0^{[n]}$, we know that the formula $x^{[n]} \times y^{[n]} \leq_u u^{[n]} \times y^{[n]}$ is valid if and only if the formula $x^{[n]} \times y^{[n]} \leq_u 0^{[n]}$ is valid. The quantified formula φ is thus equivalent with the formula $\forall x^{[n]} \forall y^{[n]} ((x^{[n]} \times y^{[n]}) \leq_u 0^{[n]})$.*

By combining ideas from Theorems 7.4, 7.5, and 7.6, we obtain the following result, which describes simplification of goal unconstrained terms in quantified formulas. Its proof is straightforward combination of the proofs of the mentioned theorems.

Theorem 7.7. *Let φ be a closed quantified formula in PNF and t be its subterm such that a subset U of unconstrained variables appearing in t satisfies $|levels(U)| = 1$ and*

$$level(U) \geq \max\left(levels(C)\right),$$

where $C = vars(t) \setminus U$. Further, let s be an arbitrary term such that and $vars(s) \cap vars(\varphi) \subseteq C$. Then the following two statements hold:

- *if all variables in U are quantified existentially, the term t occurs unsigned-antimonotonically in φ, and the terms t and s are unsigned-min C-interchangeable, the formula φ is equivalent with the formula φ where the term t is replaced by the term s and the variables in U are replaced in their quantifier block by the set of variables $vars(s) \setminus vars(\varphi)$;*
- *if all variables in U are quantified universally, the term t occurs unsigned-monotonically in φ, and the terms t and s are unsigned-min C-interchangeable, the formula φ is equivalent with the formula φ where the term t is replaced by the term s and the variables in U are replaced in their quantifier block by the set of variables $vars(s) \setminus vars(\varphi)$.*

It comes as no surprise that analogous claims hold also for unsigned-max, signed-min, and signed-max C-interchangeable terms. The only difference is that for unsigned-max and signed-max C-interchangeable terms, the monotonicity must be swapped:

- an unsigned-monotonically occurring term can be rewritten for an unsigned-max C-interchangable term if all its unconstrained variables are quantified existentially and

	Quantification of unconstrained variables	
Term occurence	Existential	Universal
unsigned-monotone	unsigned-max	unsigned-min
unsigned-antimonotone	unsigned-min	unsigned-max
signed-monotone	signed-max	signed-min
signed-antimonotone	signed-min	signed-max

Table 7.4: The table shows for each occurence of a term and each quantifier for its unconstrained variables, by which C-interchangeable term it can be replaced.

- an unsigned-antimonotonically occurring term can be rewritten for unsigned-max C-interchangable term if all its unconstrained variables are quantified universally.

All such relationships are summarized in Table 7.4.

Observe that the application of the proposed simplifications to a bit-vector term $u \circ t \bowtie s$ for a bit-vector function $\circ$ and a predicate $\bowtie$ in many cases yields precisely the corresponding invertibility condition [Nie+18b] for the formula $u \circ t \bowtie s$. However, our approach does not rely on counter-example based quantifier instantiation and can be performed as a preprocessing step in an arbitrary bit-vector SMT solver. Moreover, our approach can simplify even quantifier-free formulas and existentially quantified variables.

8 Q3B: AN EFFICIENT BDD-BASED SMT SOLVER FOR QUANTIFIED BIT-VECTORS

We have implemented all the techniques described in previous three chapters in an SMT solver called Q3B. Namely, Q3B solves satisfiability of quantified bit-vector formulas using binary decision diagrams, formula approximations, bit-vector operation abstractions, and offers simplifications of such formulas using unconstrained variables. In this chapter, we describe Q3B's architecture and the details of its implementation. The performance of the implemented solver is then experimentally evaluated in Chapter 9.

The overall approach to SMT solving used by Q3B is mostly the same as the one presented in Section 5.4. However, the approach is extended by the simplifications using unconstrained variables and all the features described in Chapter 6. In particular, the solver threads for underapproximations and overapproximations also use abstractions of bit-vector operations and refine the results as described in Section 6.3. The underapproximating and overapproximating solvers also implement checking of potential models and countermodels (Subsection 6.4.1), learning from overapproximations (Subsection 6.4.2), and introduce new variables and congruences for results of multiplications and divisions (Subsection 6.4.3). The high-level overview of thus extended approach is depicted in Figure 8.1. The refinement loop is currently using the following parameters: the initial effective bit-width is 1 and it is increased to 2, 4, 6, 8, etc.; the initial BDD node limit of bit-vector operations is 1000 and is multiplied by 4 during the refinement.

There are some differences between the theoretical description of the approach given in the previous chapters and its real implementation in Q3B. The first difference is in handling negations in the input formula. In the previous chapters, it was assumed that the input formula is in the negation normal form. However, instead of converting the input formula into the negation normal form, Q3B only eliminates bi-implications from the input formula and then maintains polarity of each subformula and uses variants of the described techniques that take the polarity into the account. For example, during the underapproximations of the formula, all existential quantifiers that occur under an odd number of negations are treated as universal quantifiers and vice-versa.

The second difference is in computing approximations of the input formula. In the theoretical description in Section 5.3, the approximations are performed by modifying the formula. In the real implementation, however, the approximations are performed during the computation of the BDD corresponding to the formula. This more closely resembles the illustration of various reductions, which is shown in Figure 5.4. In particular, vectors of BDDs that represent variables whose effective bit-width should be reduced are modified by replacing some of their BDDs either by $\boxed{0}$ or by the BDD that represents the bit of the variable that should be used as a sign bit. Therefore, some of the BDD variables

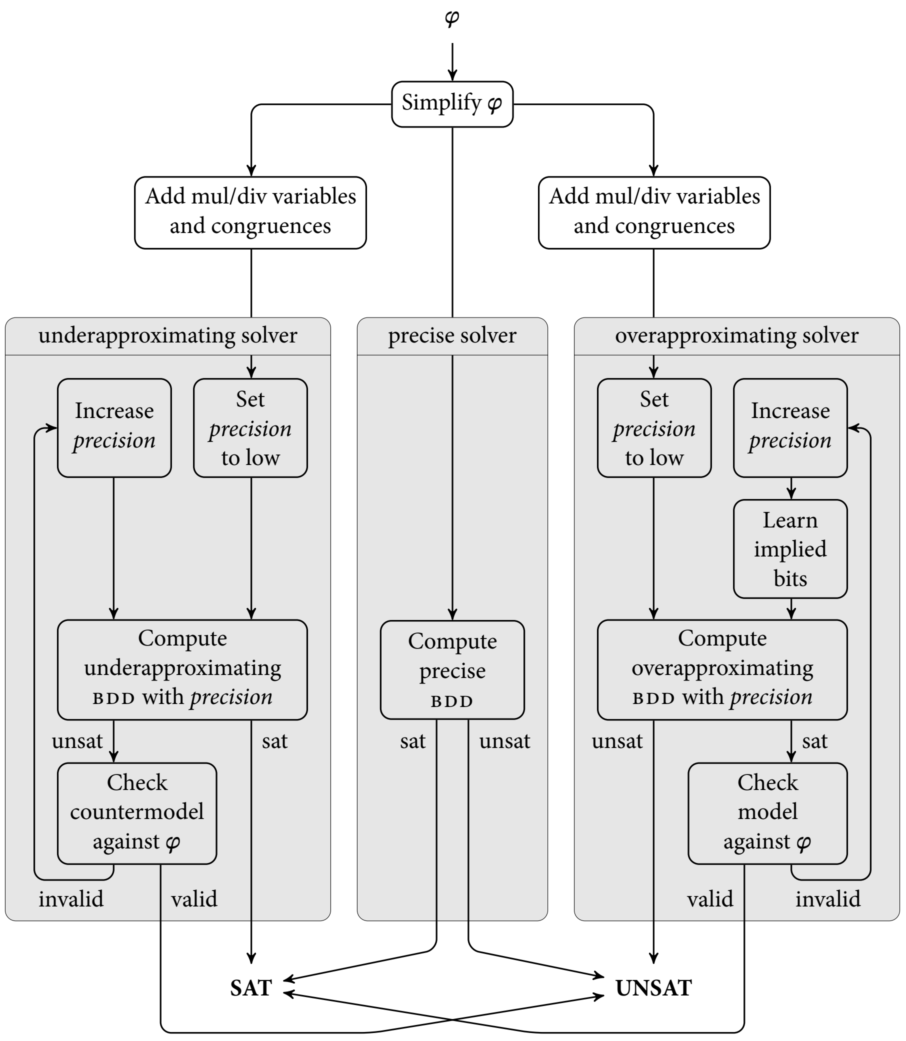


Figure 8.1: High-level overview of SMT solving approach used by Q3B. The three shaded areas are executed in parallel and the first result is returned.

that represent bits of the input bit-vector variables may not be present in the resulting BDD. The value of these bits has to be computed according to the chosen extension later-on if the model is required.

Q3B is implemented in C++17, is open-source and available under MIT license on GitHub:

https://github.com/martinjonas/Q3B.

The project development process includes continuous integration and automatic regression tests, which increase the trustworthiness of the tool.

8.1 EXTERNAL LIBRARIES

To implement all the functionality, Q3B relies on several external libraries and tools. For representation of and manipulation with BDDs, Q3B uses the open-source library CUDD 3.0 [Som15]. Since CUDD does not support bit-vector operations, we use the library by Peter Navrátil [Nav18], which on top of CUDD implements representation of bit-vectors and also implements bit-vector operations and relations. The algorithms in this library are inspired by the ones in the BDD library BuDDy[1], which provide a decent performance. Nevertheless, we have further improved its performance by several modifications. In particular, we added a specific code for handling expensive operations like bit-vector multiplication and division when their arguments contain constant BDDs. This for example considerably speeds up multiplication whenever one argument contains many constant zero bits, which is a frequent case when we use the variable bit-width approximation fixing some bits to zero. Further, we have fixed few incorrectly implemented bit-vector operations in the original library.

Moreover, to implement the abstractions of bit-vector operations, we have extended all operations in the library with the support for do-not-know bits in their inputs. We have also implemented abstract versions of arithmetic operations, so that they can produce do-not-know bits when the result exceeds a given number of BDD nodes. In particular, the operations that introduce do-not-know bits when the precise result would contain too many BDD nodes are *addition*, *multiplication*, and *division*. We have selected these operations as the original approach without operation abstractions often struggles to handle them. Finally, we have modified all algorithms for relation operators, logical operators, and quantifier processing to work with BDD pairs (b_{must}, b_{may}) rather than with individual precise BDDs.

Q3B reads the input formulas in the SMT-LIB format [BFT17], which is the standard input format for almost all modern SMT solvers. For parsing this format, Q3B uses ANTLR parser generated from the grammar[2] of SMT-LIB 2.6. We have modified the grammar to correctly handle bit-vector numerals and to support `push` and `pop` commands without numerical argument. The parser allows Q3B to support all bit-vector operations and almost all SMT-LIB commands except `get-assertions`, `get-assignment`, `get-proof`, `get-unsat-assumptions`, `get-unsat-core`, and all the commands that work with algebraic data-types. This is in sharp contrast with the initially implemented experimental versions of Q3B, which only collected all the assertions from the input file and performed the satisfiability check regardless of the rest of the commands and of the presence of the `check-sat` command. The reason for this was that the older versions parsed the input file using the C++ API of the SMT solver Z3 [MB08], which can provide only the list of assertions and not the rest of the SMT-LIB script. Thanks to the ANTLR parser, the current version of Q3B can therefore also provide the user with a model of a satisfiable formula after calling `get-model`; this important aspect of other SMT solvers was completely missing in the initial experimental versions of Q3B.

On the other hand, C++ API of the solver Z3 is still used for internal representation of parsed formulas. The Z3 C++ API is also used to perform manip-

1 https://sourceforge.net/projects/buddy/
2 https://github.com/julianthome/smtlibv2-grammar

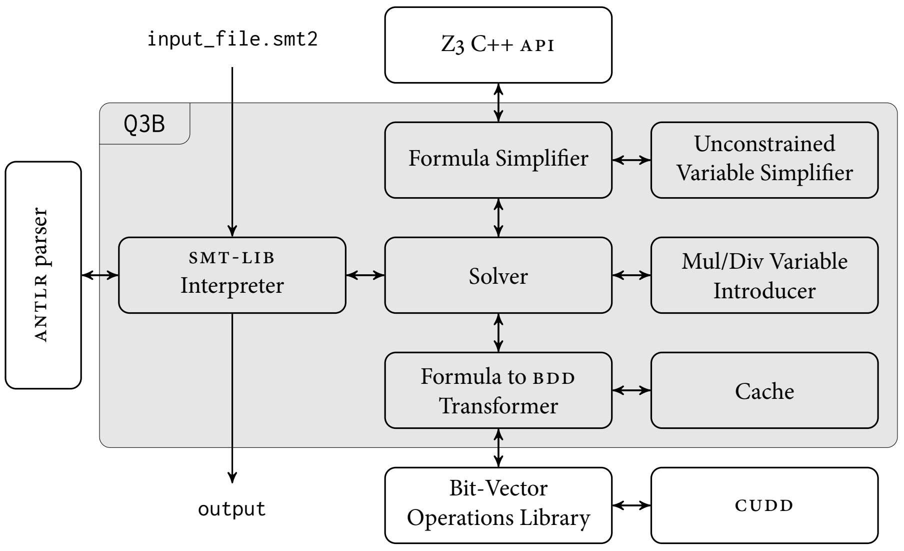

input_file.smt2
Z3 C++ API
Q3B
Formula Simplifier
Unconstrained Variable Simplifier
ANTLR parser
SMT-LIB Interpreter
Solver
Mul/Div Variable Introducer
Formula to BDD Transformer
Cache
output
Bit-Vector Operations Library
CUDD

Figure 8.2: Architecture of Q3B. Components in the shaded box are parts of Q3B, the other components are external.

ulations with formulas, such as substitution of values for variables, and some of the formula simplifications. Note that these are the only uses of Z3 API in Q3B during solving the formula; no actual SMT- or SAT-solving capabilities of Z3 are used during the solving process.

8.2 ARCHITECTURE OF Q3B

This section describes the internal architecture of Q3B. The overall structure including both internal and external components and the interactions between them is depicted in Figure 8.2. We explain the purpose of the internal components:

SMT-LIB INTERPRETER (implemented in the file `SMTLIBInterpreter.cpp`) reads the input file in SMT-LIB format. The interpreter sequentially executes all the commands from the input file. In particular, it maintains the options set by the user (commands as `set-option`, `get-option`), maintains the assertion stack (commands `assert`, `push`, and `pop`), calls the component `Solver` when the `check-sat` command is issued, and queries `Solver` if the user requires the model using the command `get-model` or its part by using `get-value`.

FORMULA SIMPLIFIER (implemented in the file `FormulaSimplifier.cpp`) provides interface to all formula simplifications described in Chapter 5. In particular, it implements miniscoping, pure literal elimination, constructive and destructive equality resolution (CER, DER), and simple theory-related rewriting. Furthermore, it also provides an interface to simplifications using unconstrained variables described in Chapter 7. Some of these simplifications are implemented directly in this component. Others, like CER, DER, and majority of the theory-related rewritings are performed by calling C++ API of Z3. Simplifications using unconstrained variables are implemented in a separate component of Q3B. The simplifier also removes top-level existential quantifiers, so values of the corresponding variables are also included in a model. This increases the chance of a successful validation of a model that is a result of an overapproximation. Some simplifications that could change models of the formula are disabled if the user enables model generation, i.e., sets the option `:produce-models` to `true` in the input SMT-LIB file.

UNCONSTRAINED VARIABLE SIMPLIFIER (implemented in the file `UnconstrainedVariableSimplifier.cpp`) provides all the simplifications of formulas with unconstrained variables that are described in Chapter 7. Besides the basic unconstrained variable simplifications, this component also provides simplifications using partially constrained terms and goal unconstrained terms.

MUL/DIV VARIABLE INTRODUCER (implemented in the file `TermConstIntroducer.cpp`) provides the functions for adding new variables for results of multiplications and divisions and also for adding their congruences, as described in Subsection 6.4.3.

SOLVER (implemented in `Solver.cpp`) is the central component of our tool. It calls formula simplifier and then creates three threads for the precise solver, the underapproximating solver, and the overapproximating solver. It also controls the approximation refinement loops of the approximating solvers. Finally, it returns the result of the fastest thread and stores the respective model, if the result was `sat`.

FORMULA TO BDD TRANSFORMER (implemented in the source file `ExprToBDDTransformer.cpp`) performs the actual conversion of the formula to a BDD. Each subterm of the input formula is converted to a vector of BDDs (if the subterm's sort is a bit-vector of width n, the constructed vector contains n BDDs, each of whose represents one bit of the subterm). Further, each subformula of the input formula is converted to a corresponding BDD. These conversions proceed by a straightforward bottom-up recursion on the formula syntax tree. The transformer component calls an external library to compute the effect of logical and bit-vector operations on BDDs and vectors of BDDs, respectively. Besides the precise conversion, the transformer can also construct overapproximating and underapproximating BDDs or the pairs of BDDs (b_{must}, b_{may}). Precision of approximations depends on parameters set by the solver component. The solver component also computes the heuristic initial ordering of the BDD variables as described in Section 5.2.

CACHE (implemented as a part of `ExprToBDDTransformer.cpp`) maintains for each converted subformula and subterm the corresponding BDD or a vector of BDDs, respectively. Each of the three solvers has its own cache.

When an approximating solver increases the precision of the approximation, only entries of its cache that can be affected by the precision change are invalidated. This ensures that results for subterms and subformulas that were not affected by the refinement do not have to be recomputed in each iteration of the refinement loop. All the caches are internally implemented by hash-tables.

Some of the components also provide the public C++ API, which can be used by external tools for SMT solving or for performing formula simplifications. In particular, this is the of with components `FormulaSimplifier`, `UnconstrainedVariableSimplifier`, and `Solver`. For example, `Solver` exposes a method `Solve(formula, approximationType)`, which can be used to decide satisfiability of a given formula by performing simplifications and running the precise solver, the underapproximating solver, or the overapproximating solver. The component `Solver` also exposes the method `SolveParallel(formula)`, which simplifies the input formula and runs all three of these solvers in parallel and returns the first result as depicted in Figure 8.1.

8.3 FUTURE WORK

In future, we would like to drop the dependency on Z3: namely to implement our own representation of formulas and reimplement all the simplifications currently outsourced to Z3 API directly in Q3B. We also plan to extend some simplifications with an additional bookkeeping needed to construct a model of the original formula. With these extensions, all simplifications could be used even if the user wants to get a model of the formula. As a further research topic, we would also like to implement production of unsatisfiable cores since they are also valuable for software verification.

We would also like to experiment with different decision diagrams than BDDs. Namely, we would like to try Chain-reduced BDDs [Bry18], Tagged BDDs [DWM17], or BDDs with Edge-Specified Reductions [Bab+19], which should combine benefits of BDDs and Zero-suppressed Binary Decision Diagrams [Min01]. As a preliminary investigation of these decision diagrams, we have already implemented a version of Q3B that uses Chain-reduced BDDs. In theory, these should provide smaller representation of BDDs such as the one in Figure 5.2 because the variables x_0, x_1, x_2, and x_3 can be represented by a single node. However, this did not happen in practice as the variables x_0, x_1, x_2, and x_3 do not form a consecutive block because they are interleaved with variables v_0, v_1, v_2, and v_3. We therefore propose further investigation of a compatible variable ordering or improved variants of Chain-reduced BDDs, which would allow representing blocks of non-consecutive variables.

9 EXPERIMENTAL EVALUATION OF Q3B

This chapter experimentally evaluates the performance of the implemented SMT solver Q3B. Section 9.1 compares the performance of Q3B against other state-of-the-art SMT solvers for quantified bit-vector formulas. Section 9.2 investigates the effect of the individual techniques used by Q3B on its performance.

As the benchmark set, we have used all 5751 quantified bit-vector formulas from the SMT-LIB repository. The benchmarks are divided into 8 distinct families of formulas:

- 5 families of formulas *2017-Preiner-**, which were obtained by M. Preiner by conversion of integer and real arithmetic benchmarks to bit-vectors of bit-width 32 [PNB17].

  For brevity, we call the family 2017-Preiner-UltimateAutomizer *as* 2017-Preiner-ua.

- The family *2018-Preiner-cav*, which consists of formulas that validate the generated invertibility condition formulas that are used in the SMT solver CVC4 [Nie+18b].

- The family *heizmann-ua*, which contains benchmarks from the tool Ultimate Automizer by M. Heizmann [HHP13].

  The original name of this family is 20170501-Heizmann-UltimateAutomizer; *we use the shorter version for brevity.*

- The family *wintersteiger*, which contains benchmarks from software and hardware verification by C. M. Wintersteiger [WHM13].

All benchmarks were executed with CPU time limit 20 minutes and RAM limit of 16 GiB. All experiments were performed on a Debian machine with two six-core Intel Xeon E5-2620 2.00GHz processors and 128 GB of RAM. All the measured times are CPU times, unless stated otherwise. For reliable benchmarking we employed BenchExec [BLW15], a tool that allocates specified resources for a program execution and precisely measures their usage. All scripts used for running benchmarks and processing their results, together with detailed descriptions and some additional results not presented in the chapter, are available online[1].

9.1 OVERALL PERFORMANCE

We have evaluated the performance of QB3 1.0 and compared it to the latest versions of SMT solvers Boolector (v3.0), CVC4 (v1.6), and Z3 (v4.8.4). All tools were used with their default settings except for CVC4, where we used the same settings as in the paper that introduces quantified bit-vector solving in CVC4 [Nie+18b], since they give better results than the default CVC4 settings.

1 https://fi.muni.cz/~xjonas/PhdThesis/Q3BExperiments

	Family	Total	Boolector	CVC4	Q3B	Z3
UNSAT	2017-Preiner-keymaera	3925	3916	3919	3905	**3923**
	2017-Preiner-psyco	62	**62**	61	59	**62**
	2017-Preiner-scholl-smt08	75	62	34	**70**	68
	2017-Preiner-tptp	56	53	**56**	**56**	**56**
	2017-Preiner-ua	137	136	**137**	**137**	**137**
	2018-Preiner-cav18	590	549	**565**	**565**	550
	heizmann-ua	110	11	**109**	104	17
	wintersteiger	94	85	89	**92**	**92**
	Total UNSAT	5049	4874	4970	**4988**	4905
SAT	2017-Preiner-keymaera	108	103	77	104	**108**
	2017-Preiner-psyco	132	131	129	123	**132**
	2017-Preiner-scholl-smt08	256	244	214	**247**	204
	2017-Preiner-tptp	17	16	**17**	**17**	**17**
	2017-Preiner-ua	16	**16**	14	**16**	**16**
	heizmann-ua	21	19	19	**20**	15
	wintersteiger	91	77	85	**90**	71
	Total SAT	641	606	555	**617**	563
	Total	5751	5480	5525	**5605**	5468

Table 9.1: Numbers of benchmarks solved by the individual solvers divided by the satisfiability/unsatisfiability and the benchmark family.

	Boolector	CVC4	Q3B	Z3	Uniquely solved
Boolector	0	118	68	72	7
CVC4	163	0	61	173	6
Q3B	193	141	0	206	25
Z3	60	116	69	0	7

Table 9.2: The table shows cross-comparison of solved benchmarks for all pairs of the solvers. Each cell shows the number of benchmarks that were solved by the solver in the corresponding row, but not by the solver in the corresponding column. The column *Uniquely solved* shows the number of benchmarks that were solved only by the given solver.

family	Boolector	CVC4	Q3B	Z3
2017-Preiner-keymaera	1	0	1	4
2017-Preiner-psyco	0	0	0	1
2017-Preiner-scholl-smt08	5	2	10	1
2018-Preiner-cav18	1	0	11	0
heizmann-ua	0	3	1	0
wintersteiger	0	1	2	1

Table 9.3: For each solver and benchmark family, the table shows the number of benchmarks solved only by the given solver.

Table 9.1 shows the numbers of benchmarks solved by the individual solvers. The table is divided according to the satisfiability of the benchmarks and according to their families. Our solver Q3B is able to solve the most benchmarks in benchmark families *2017-Preiner-scholl-smt08*, *2017-Preiner-tptp*, *2017-Preiner-ua*, *2018-Preiner-cav18*, and *wintersteiger*, and it is competitive in the remaining families. In total, Q3B also solves more formulas than each of the other solvers: 125 more than Boolector, 80 more than CVC4, and 137 more than Z3. Although the numbers of solved formulas for the solvers seem fairly similar, the cross-comparison in Table 9.2 shows that the differences among the individual solvers are actually larger. For each other solver, there are at least 141 benchmarks that can be solved by Q3B but not by the other solver. We think this shows the importance of developing an SMT based on BDDs, approximations, and abstractions besides the solvers based on quantifier instantiation. For completeness, Table 9.3 shows the distribution of the uniquely solved benchmarks between the benchmark families.

From the opposite point of view, Figure 9.1 shows the number of benchmarks *unsolved* by the individual solvers. This plot graphically shows that Q3B solves substantially more benchmarks than the other solvers and that its performance is quite stable across the benchmark families.

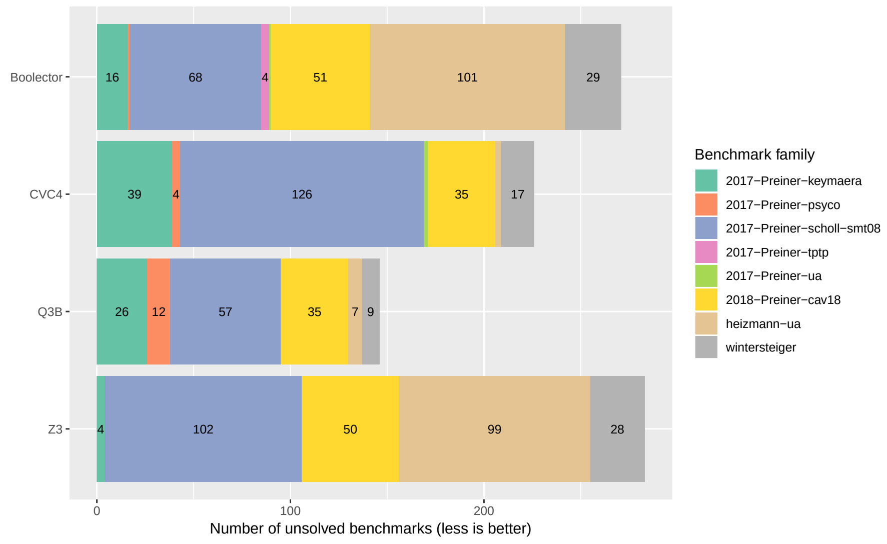


Figure 9.1: The number of benchmarks unsolved by the individual solvers. The benchmarks are divided by the source of the benchmark. For better readability, the numbers of unsolved benchmarks less than 3 are not explicitly spelled out in the plot.

As the solving times are concerned, Figure 9.2 shows quantile plots of solving times of all the compared solvers. The plot shows that Q3B performs worse on the easier benchmarks but as the benchmarks grow harder, it tends to outperform the other solvers. To see the differences more clearly, Figure 9.3 shows the same plot with only *non-trivial* benchmarks. In this plot, we have filtered out 3225 trivial benchmarks, i.e., the benchmarks that were decided by all of the solvers in less than 0.1 s.

9.2 EFFECT OF INDIVIDUAL TECHNIQUES

This section evaluates the effect of the individual introduced techniques on the performance if Q3B. In particular, we test the following three base configurations of Q3B:

- `full`, which uses both variable bit-width approximations and abstractions of bit-vector operations (the default configuration of Q3B),
- `approx`, which uses only variable bit-width approximations (command-line option `--abstract:method=variables`), and
- `none`, which performs only simplifications and conversion to the corresponding BDD without any approximations or abstractions (command-line option `--abstractions=none`).

For each base configuration *conf* of these three, we have evaluated its three variants:

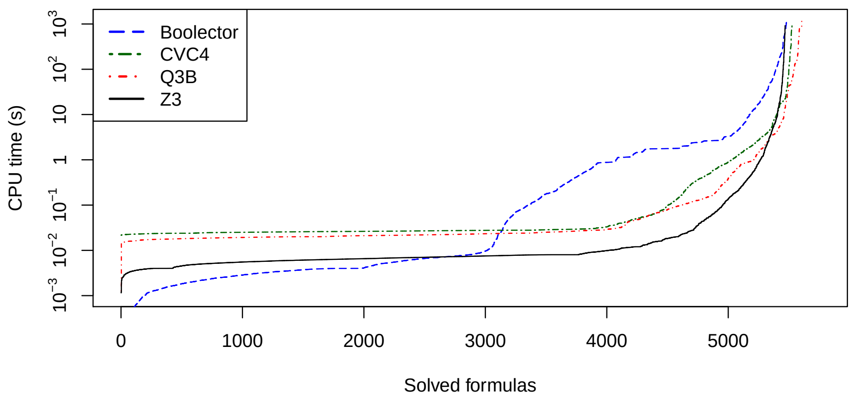


Figure 9.2: Quantile plot of all benchmarks from the SMT-LIB repository. The plot shows the number of non-trivial benchmarks (x-axis) that each solver was able to decide within a given CPU time limit (y-axis).

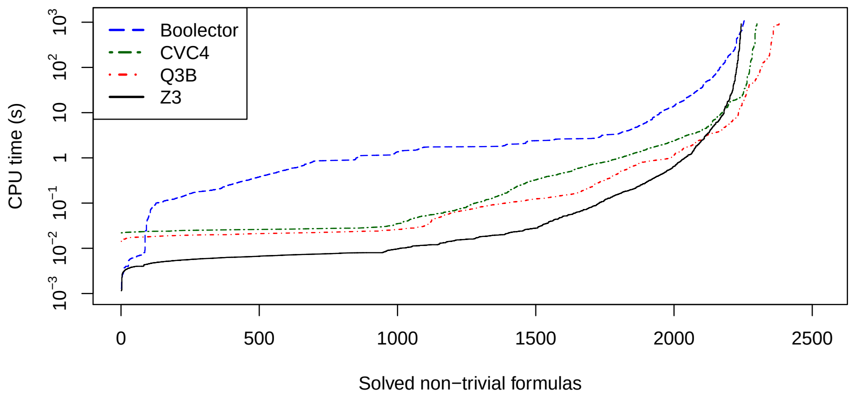


Figure 9.3: Quantile plot of all solved non-trivial benchmarks from the SMT-LIB repository. Trivial benchmarks are those that all solvers solved within 0.1 s. The plot shows the number of non-trivial benchmarks (x-axis) that each solver was able to decide within a given CPU time limit (y-axis).

- *conf*-gu, which uses simplifications of unconstrained variables in unconstrained terms, partially unconstrained terms, and goal unconstrained terms (the default configuration of Q3B),
- *conf*-u, which uses simplifications of unconstrained variables only in unconstrained terms and partially unconstrained terms (command-line option `--uc:goal=0`), and
- *conf* without no suffix, which uses no simplifications of unconstrained variables (command-line option `--simpl:unconstrained=0`).

In total, this gives us 9 different configurations of Q3B. We have run each of these configurations on all 5751 described benchmarks with 20 minutes of CPU time limit and 16 GiB of RAM.

Figure 9.4 shows the number of solved satisfiable formulas, solved unsatisfiable formulas, and unsolved formulas for each of the 9 described configurations. It can be seen that both approximations of bit-width and abstractions of bit-vector operations significantly improve the number of solved formulas. Similarly, both simplification of unconstrained terms and of goal unconstrained terms also improve the performance of the solver. Furthermore, these simplifications improve the performance regardless the used approximations or abstractions. On the other hand, approximations bring the additional cost of running three parallel solvers and abstractions bring the additional cost of the repeated abstraction refinement. Therefore, there are benchmarks where approximations and abstractions degrade the performance.

Further, we compare the various configurations of Q3B against the state-of-the-art SMT solvers Boolector, CVC4, and Z3. To compare the numbers in Figure 9.4 with the numbers for solvers, note that the number of benchmarks unsolved by Boolector, CVC4, and Z3, are 271, 226, and 283, respectively. Even the configuration `full-u` without the simplifications of goal unconstrained terms can solve more formulas than these SMT solvers. Furthermore, configurations `full`, `approx-gu` and `approx-u` are competitive with these SMT solvers. Observe that the performance of the configurations `none-*` is significantly worse. However, this is expected since these configurations time-out for most of the benchmarks that contain multiplication of two non-constant terms because it leads to an exponential BDD.

For the selected configurations of Q3B, Table 9.4 shows the numbers of solved benchmarks grouped by their satisfiability and family. From the opposite view, the number of *unsolved* benchmarks grouped by their family is shown in Figure 9.5 for all of the configurations of Q3B. In particular, it can be observed that effective bit-width approximations are particularly helpful for the following benchmark families: `2017-Preiner-keymaera`, `2017-Preiner-psycho`, `2017-Preiner-scholl-smt08`, and `wintersteiger`. Additionally, abstractions of bit-vector operations are particularly helpful for families `2017-Preiner-keymaera` and `2017-Preiner-scholl-smt08`.

On the other hand, simplification using unconstrained and partially unconstrained terms are helpful for categories `heizmann-ua` and `2018-Preiner-cav18`. This is not surprising as the benchmarks `heizmann-ua` come from soft-

	Family	Total	`none`	`approx`	`approx-u`	`full`	`full-u`	`full-gu`
UNSAT	2017-Preiner-keymaera	3905	3781	3783	3786	3905	3905	3905
	2017-Preiner-psyco	59	44	55	56	58	59	59
	2017-Preiner-scholl-smt08	71	36	44	59	65	**70**	**70**
	2017-Preiner-tptp	56	54	**56**	**56**	**56**	**56**	**56**
	2017-Preiner-ua	137	**137**	**137**	**137**	**137**	**137**	**137**
	2018-Preiner-cav18	532	435	448	531	453	530	**565**
	heizmann-ua	107	85	80	104	82	104	104
	wintersteiger	93	75	**92**	**92**	**92**	**92**	**92**
	Total UNSAT	4960	4647	4695	4821	4848	4953	**4988**
SAT	2017-Preiner-keymaera	104	7	103	104	104	104	104
	2017-Preiner-psyco	126	74	124	124	123	123	123
	2017-Preiner-scholl-smt08	248	136	226	229	**246**	**247**	**247**
	2017-Preiner-tptp	17	13	**17**	**17**	**17**	**17**	**17**
	2017-Preiner-ua	16	**16**	**16**	**16**	**16**	**16**	**16**
	heizmann-ua	20	18	18	18	**20**	**20**	**20**
	wintersteiger	92	55	**92**	**92**	**90**	**90**	**90**
	Total SAT	623	319	596	600	**616**	**617**	**617**
	Total	5751	4966	5291	5421	5464	**5570**	**5605**

Table 9.4: Numbers of benchmarks solved by the individual Q3B configurations grouped by the satisfiability/unsatisfiability and the benchmark family. The families in which the given configuration solved more benchmarks than Boolector, CVC4, and Z3 are marked **bold**.

ware verification problems, which tend to contain unconstrained variables, and all benchmarks in the family `2018-Preiner-cav18` contain unconstrained variables by construction. For the family `heizmann-cav18`, the simplification using goal unconstrained terms help even further.

Interestingly, the most basic configuration `none` of Q3B, which uses only simplifications and conversion to BDD can solve significantly more unsatisfiable benchmarks from category `heizmann-ua` than Boolector and Z3.

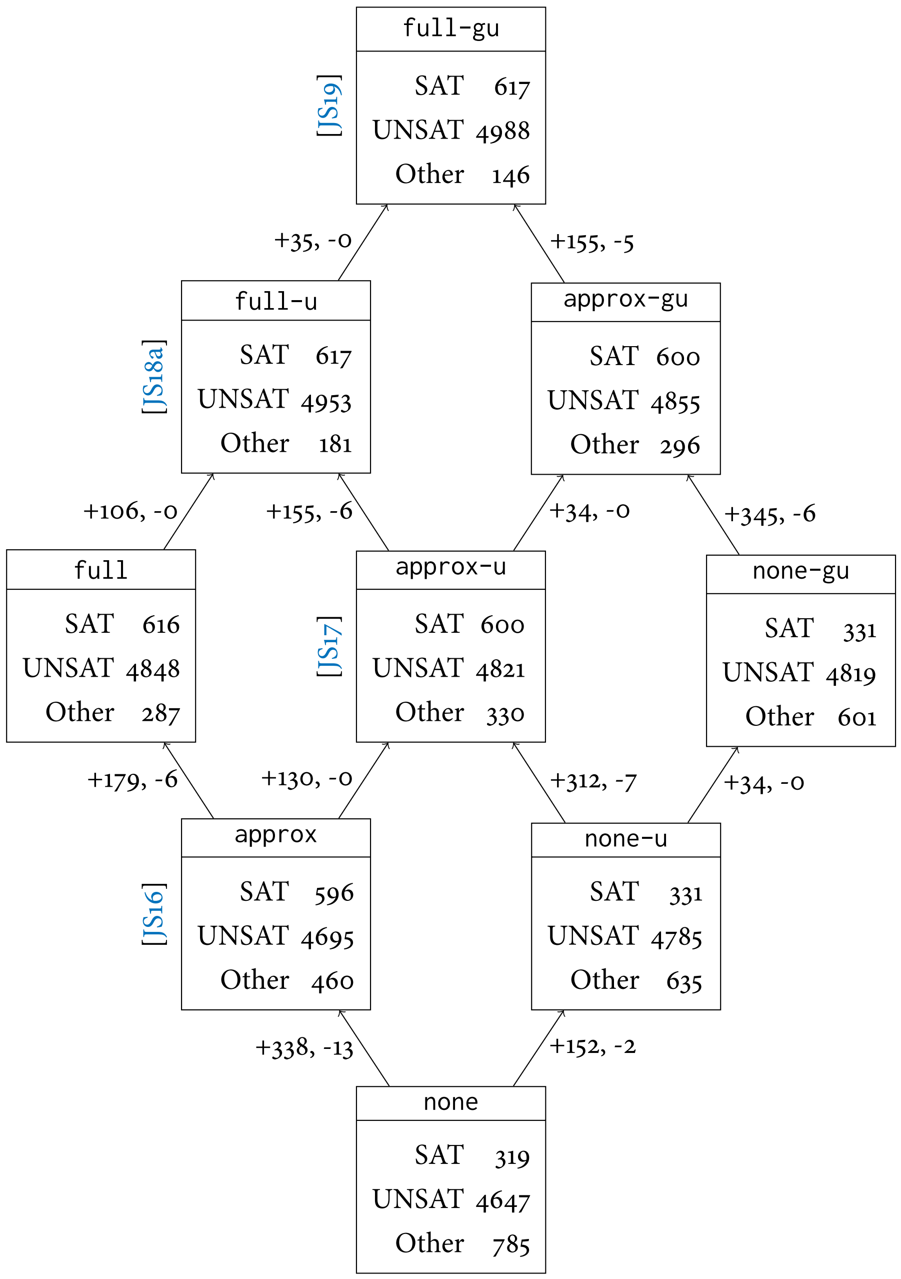


Figure 9.4: Numbers of solved satisfiable formulas, solved unsatisfiable formulas, and unsolved formulas by the individual configurations of Q3B. If a configuration was used in a paper, it is referenced besides the corresponding configuration. Each arrow from configuration c_1 to c_2 labeled `+x, -y` denotes that the configuration c_2 solved x formulas that c_1 did not and that the configuration c_2 did not solve y formulas that c_1 did.

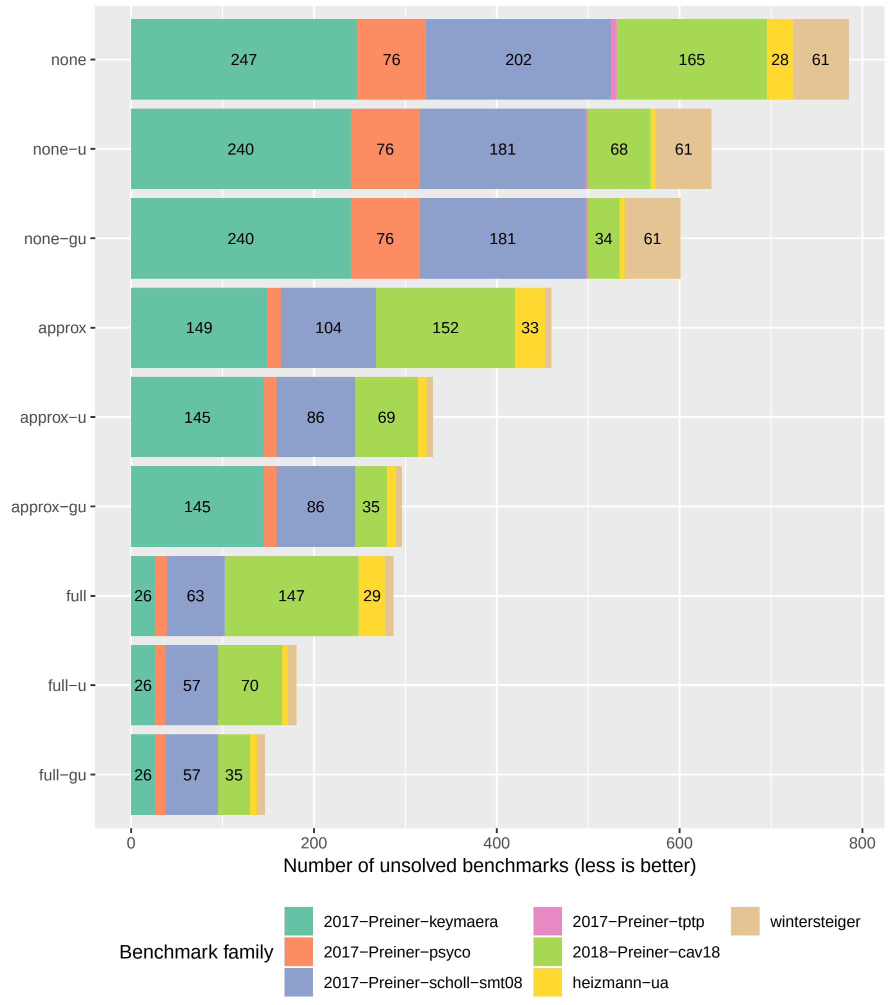


Figure 9.5: The number of benchmarks unsolved by the individual configurations of Q3B. The benchmarks are grouped by the source of the benchmark. For better readability, the numbers of unsolved benchmarks less than 20 are not explicitly spelled out in the plot.

10 IS SATISFIABILITY OF QUANTIFIED BIT-VECTOR FORMULAS STABLE UNDER BIT-WIDTH CHANGES?

For most of the decision procedures for solving satisfiability of *quantifier-free* bit-vector formulas, their time and space complexities grow with the increasing bit-widths of used variables. This is caused by the conversion of the input formula to the equisatisfiable propositional formula (*bit-blasting*) and solving it by a SAT solver, which is either the main or a fall-back strategy for most of the current decision procedures. The problem with growing bit-widths in turn applies to decision procedures for *quantified* bit-vectors based on the quantifier-instantiation used by Boolector [NPB17], CVC4 [Nie+18b], and Z3 [WHM13], which employ a solver for quantifier-free formulas as a black box. The approach to quantified bit-vector formulas based on binary decision diagrams used by the solver Q3B is sensitive to bit-widths as well: as the bit-widths increase, so do the sizes of the produced binary decision diagrams. These claims are supported by plots in Figure 10.1, which show solving times of the solver CVC4 on almost all 32bit and 64bit quantified bit-vector formulas from the SMT-LIB repository [BFT16] after reducing their bit-widths to all values between 1 and the original bit-width. Details concerning the selection of benchmarks and reduction of their bit-widths are described in Section 10.1.

Under reasonable complexity-theory assumptions, the effect of increasing bit-widths is also observable from the complexity-theory point of view based on the results mentioned in Chapter 4. For both quantifier-free and quantified formulas, the respective complexity classes of deciding satisfiability of formulas with bit-widths encoded in unary and in binary differ: **NP** vs. **NEXPTIME** for quantifier-free formulas and **PSPACE** vs. **AEXP**(poly) for quantified formulas. Therefore, the decision problem has to become harder with the increas-

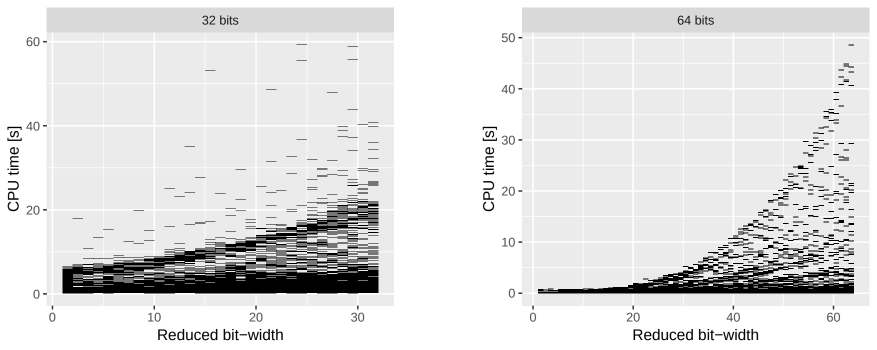


Figure 10.1: The scatter plots show CPU times of the solver CVC4 on a subset of 32bit and 64bit formulas from the SMT-LIB repository after changing their bit-widths to the bit-width specified on the x-axis. The formulas are divided according to the maximal bit-width of their subterms.

ing bit-widths, otherwise the complexity classes for unary and binary encoding would coincide.

For quantifier-free bit-vector formulas, this problem has been tackled several times. As was already mentioned in Section 5.3, Bryant et al. have proposed an abstraction-based procedure that tries to solve underapproximations of the input formula with reduced effective bit-widths of variables and selectively increases their bit-widths if the underapproximation is unsatisfiable. If the underapproximation is satisfiable, the original formula can be decided to be satisfiable as well [Bry+07]. Approaching the problem from another side, Fröhlich et al. have proposed stochastic local search approach that randomly looks for models of quantifier-free formulas and thus avoids bit-blasting with its inherent space-complexity dependence on the bit-widths of the input formula [Frö+15]. This approach was improved by propagation rules by Niemetz et al. [NPB17]. Zeljić et al. have developed the solver mcBV [ZWR16], which implements the model-constructing satisfiability calculus [MJ13] that tries to construct a model directly and thus also avoids bit-blasting. Johannsen has shown how to compute a bit-width to which a formula can be reduced while preserving its satisfiability for a restricted class of formulas that represent *bit-wise functions* [Joh01; Joh02]. Kovásznai et al. used this observation regarding such formulas to show that the complexity of their satisfiability is the same for both unary and binary encoding of the bit-widths [KFB16]. Considering a related theory of *floating-point arithmetic*, Zeljić et al. have introduced an approximation framework that produces mixed-approximations of the original formula by reducing the bit-widths used for representation of all floating-point variables. After solving the mixed-approximation, the solver checks its result against the original formula and if it fails, it refines the approximation [ZWR17; Zel+18]. In general, this approach works for an arbitrary quantifier-free theory, but as far as we know, it has been implemented only for floating-point arithmetic.

For quantified bit-vector formulas, only one existing SMT solver tries to reduce bit-widths of some variables of the input formula: the solver Q3B uses approximations inspired by the abstractions by Bryant et al. and computes underapproximations and overapproximations by reducing bit-widths of existentially and universally quantified variables, respectively. This is discussed in detail in Section 5.3.

In this chapter, we experimentally evaluate the hypothesis that the satisfiability of only a small fraction of quantified bit-vector formulas actually depends on the bit-widths of the used variables. We show that the satisfiability of the vast majority of quantified bit-vector formulas from the SMT-LIB repository remains the same even after reducing bit-widths of their variables to a very small number of bits. Therefore, the results of this chapter suggest that extending the techniques described for quantifier-free formulas to quantified formulas or designing novel techniques for reducing bit-widths of quantified bit-vector formulas could be worthwhile. These techniques could allow SMT solvers both to solve more quantified bit-vector formulas and to solve them more quickly.

The chapter is structured as follows: Section 10.1 describes how we obtain formulas with reduced bit-widths, Section 10.2 presents our experiments with the reduced formulas and results of these experiments, and the following Section 10.3 discusses challenges arising from these results.

10.1 OBTAINING FORMULAS WITH REDUCED BIT-WIDTHS

This section describes how we obtain formulas with the reduced bit-width from the original benchmarks. For each term, we denote as $\mathrm{bw}(t)$ its bit-width. Given a formula φ and a desired bit-width $bw \geq 1$, the following procedure produces the formula $\mathit{reduceF}(\varphi, bw)$ in which all subterms have bit-width at most bw. In particular, $\mathit{reduceF}(\varphi, bw)$ is obtained from φ by:

- decreasing bit-widths of all variables with the bit-width more than bw to bw,
- replacing all numerals with the bit-width more than bw by their bw least-significant bits,
- decreasing the numbers of added bits by all `zeroExtend` and `signExtend` functions so that the bit-width of the result is at most bw.

Formally, we introduce the two mutually recursive functions *reduceT* and *reduceF*. The first one, *reduceT*, performs the above-described reduction of maximal bit-width on bit-vector terms:

$$\begin{aligned}
\mathit{reduceT}(c^{[n]}, bw) &= \left(c \bmod 2^{\min(n,bw)}\right)^{[\min(n,bw)]},\\
\mathit{reduceT}(x^{[n]}, bw) &= x^{[\min(n,bw)]},\\
\mathit{reduceT}(op(t_1), bw) &= op(\mathit{reduceT}(t_1, bw))\\
&\quad \text{for } op \in \{-, \sim\},\\
\mathit{reduceT}(t_1 \diamond t_2, bw) &= \mathit{reduceT}(t_1, bw) \diamond \mathit{reduceT}(t_2, bw)\\
&\quad \text{for } \diamond \in \{\&, |, +, \times, /_u, /_s, \%_u, \%_s, \ll, \gg_u, \gg_s\},\\
\mathit{reduceT}(\mathtt{ite}(\varphi, t_1, t_2), bw) &= \mathtt{ite}(\mathit{reduceF}(\varphi, bw),\\
&\qquad \mathit{reduceT}(t_1, bw),\\
&\qquad \mathit{reduceT}(t_2, bw)),\\
\mathit{reduceT}(ext_n(t), bw) &= \begin{cases} \mathit{reduceT}(t, bw) & \text{if } \mathrm{bw}(t) \geq bw,\\ ext_{\min(n, bw-\mathrm{bw}(t))}(t) & \text{if } \mathrm{bw}(t) < bw, \end{cases}\\
&\quad \text{for } ext \in \{\mathtt{zeroExtend}, \mathtt{signExtend}\}.
\end{aligned}$$

By using the function *reduceT* on arguments of relation symbols in the formula, we obtain the function *reduceF*, which reduces maximal bit-widths in arbitrary formulas:

$$\begin{aligned}
\mathit{reduceF}(\top, bw) &= \top, \\
\mathit{reduceF}(\bot, bw) &= \bot, \\
\mathit{reduceF}(t_1 \bowtie t_2, bw) &= \mathit{reduceT}(t_1, bw) \bowtie \mathit{reduceT}(t_2, bw) \\
& \text{for } \bowtie \in \{=, \leq_u, <_u, \leq_s, <_s\}, \\
\mathit{reduceF}(\neg\varphi, bw) &= \neg \mathit{reduceF}(\varphi, bw), \\
\mathit{reduceF}(\varphi_1 \diamond \varphi_2, bw) &= \mathit{reduceF}(\varphi_1, bw) \diamond \mathit{reduceF}(\varphi_2, bw) \\
& \text{for } \diamond \in \{\wedge, \vee\}, \\
\mathit{reduceF}(Qx^{[n]}\,(\varphi), bw) &= Qx^{[\min(n, bw)]}\,(\mathit{reduceF}(\varphi, bw)) \\
& \text{for } Q \in \{\forall, \exists\}.
\end{aligned}$$

Note that the function *reduceT* is undefined on terms that contain extraction or concatenation. We have decided to exclude formulas that contain these operations for the following reasons. For extraction, there are multiple arbitrary choices of bits to extract; e.g., the extraction of the middle (i.e. the third) bit from a 5bit variable reduced to 3 bits could extract the middle (i.e. the second) bit or the third bit. Reduction of a formula containing concatenation may require reducing a single variable to multiple different bit-widths – although this is possible to achieve by adding extractions, this would change the semantics of the formula beyond merely changing the bit-widths of variables in which we are interested. For example, after reducing the formula

$$\mathtt{concat}(x^{[4]}, y^{[4]}) = \mathtt{concat}(y^{[4]}, x^{[4]})$$

to 6 bits, the variable x would get reduced to 2 bits on the left-hand side, but would stay 4bit on the right-hand side.

10.1.1 *Resulting Reduced Formulas*

We have written a simple tool that for an input formula φ in the SMT-LIB format [BFT17] generates formulas *reduceF*(φ, i) for all i between 1 and the maximal bit-width (included) of any subterm of the formula φ. The tool uses API of the SMT solver Z3 [MB08] and it is available at

`https://gitlab.fi.muni.cz/xjonas/FormulaReducer`.

Using this tool, we have generated reduced versions of all quantified bit-vector formulas from the SMT-LIB repository except for

- all 400 4bit and 32bit formulas from the *2018-Preiner-cav18* benchmark family. This does not lead to loss of information because all these formulas are generated from the remaining 64bit formulas by the function *reduceF* with the parameter 4 and 32, respectively;

Benchmark family	Total	bw > 100	concat extract	T/O	Remaining
2017-Preiner-keymaera	4035	0	0	65	3970
2017-Preiner-psyco	194	0	0	5	189
2017-Preiner-scholl-smt08	374	0	0	153	221
2017-Preiner-tptp	73	0	0	0	73
2017-Preiner-ua	153	0	0	2	151
2018-Preiner-cav18	200	0	40	30	130
heizmann-ua	131	0	4	7	120
wintersteiger	191	21	67	52	51
Total	5351	21	111	314	4905

Table 10.1: The table shows the formulas excluded from our evaluation according to their families. The column *Total* shows a total number of formulas in each family (except for *2018-Preiner-cav18*, where all 4bit and 32bit benchmarks have been excluded). Next three columns show numbers of formulas excluded because of too large bit-width, use of operations `concat` or `extract`, and timeout of the solver for any of the reduced versions of the formula, respectively. The last column shows the number of remaining formulas on which our evaluation of effects of bit-width reduction on satisfiability was performed.

- 21 formulas that contain subterms of bit-width larger than 100 to keep the solving time reasonable;
- 111 formulas that use operations `concat` or `extract`.

Table 10.1 shows the numbers of such excluded benchmarks according to their families.

From the set of 5219 non-excluded original formulas, we have generated in total 173 105 corresponding formulas with the reduced bit-widths. An archive containing all these generated formulas can be found at

`http://fi.muni.cz/~xstrejc/lpar2018/ReducedBW.tar.gz`.

10.2 EXPERIMENTAL EVALUATION

We have evaluated satisfiability of all the resulting 173 105 formulas. For the evaluation, we have used the SMT solver CVC4 [Nie+18b], as it is the winner of the SMT Competition 2018 in the category of quantified bit-vector formulas. The solver was run with 1 minute CPU time limit and 8 GiB RAM limit. For this, we again employed BenchExec [BLW15]. All experiments were performed on a Debian machine with two six-core Intel Xeon E5-2620 2.00GHz processors and 128 GB of RAM.

From the original 5219 formulas, 4905 were decided by CVC4 for all bit-widths. On the other hand, CVC4 exceeded the time limit on at least one bit-

Benchmark family	Benchmarks	≥ 1b	≥ 2b	≥ 4b	≥ 8b
2017-Preiner-keymaera	3970	64	19	12	4
2017-Preiner-psyco	189	49	5	0	0
2017-Preiner-scholl-smt08	221	1	0	0	0
2017-Preiner-tptp	73	20	10	3	0
2017-Preiner-ua	151	41	32	3	0
2018-Preiner-cav18	130	4	3	3	3
heizmann-ua	120	29	20	6	5
wintersteiger	51	8	6	5	2
Total	4905	216	95	32	14
%		4.4	1.9	0.65	0.29

Table 10.2: The table shows the numbers of benchmarks in the individual families whose satisfiability status is different for the original formula and for any reduced formula with bit-width at least 1, 2, 4, and 8 bits, respectively.

width on the remaining 314 formulas. The distribution of these non-decided formulas among benchmark families can be found in Table 10.1. We excluded these 314 formulas from the evaluation and performed the evaluation only on 4905 formulas with known status for all bit-widths. When grouped by their maximal bit-width, the set of evaluated formulas contains 10 formulas of bit-width 1; 10 of bit-width 8; 20 of bit-width 20; 4727 of bit-width 32; 1 of bit-width 33; 134 of bit-width 64; and 3 of bit-width 65.

10.2.1 *Satisfiability of Formulas with Reduced Bit-Widths*

Surprisingly, from the 4905 formulas, only 4.4% have a different satisfiability status for the original formula and any of its reduced version. Moreover, only 1.9% have a different satisfiability status after reducing to 2 bits or more, 0.65% have a different satisfiability status after reducing to 4 bits or more, and only 0.29% have a different satisfiability status after reducing to 8 bits or more. Table 10.2 shows the numbers of such benchmarks precisely after grouping the formulas to their respective families. For example, the table shows that all decided formulas from the family *2017-Preiner-scholl-smt08*, which contains the largest number of undecided benchmarks, have the same satisfiability status for all bit-widths from 2 to the original bit-width and all decided formulas from the family *2017-Preiner-psyco* have the same satisfiability status for all bit-widths from 4 to the original bit-width. Note that the original bit-width of all benchmarks from these families is 32 bits.

Figure 10.2 presents these results graphically for all bit-widths between 1 and the original bit-width. The figure shows the results only for formulas with the original bit-width of 32 or 64 bits as the number of benchmarks of other bit-widths is negligible. The figure for example shows that only under 0.25% of

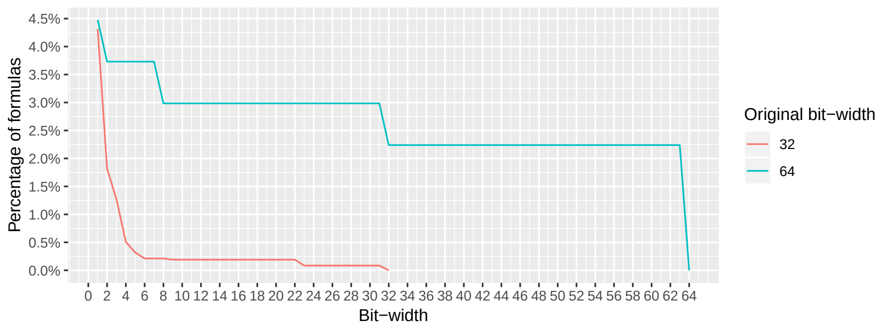


Figure 10.2: The plot shows the percentage of 32bit and 64bit benchmarks (y-axis) whose satisfiability status is different for the original formula and for any reduced formula with a given (x-axis) or a larger bit-width.

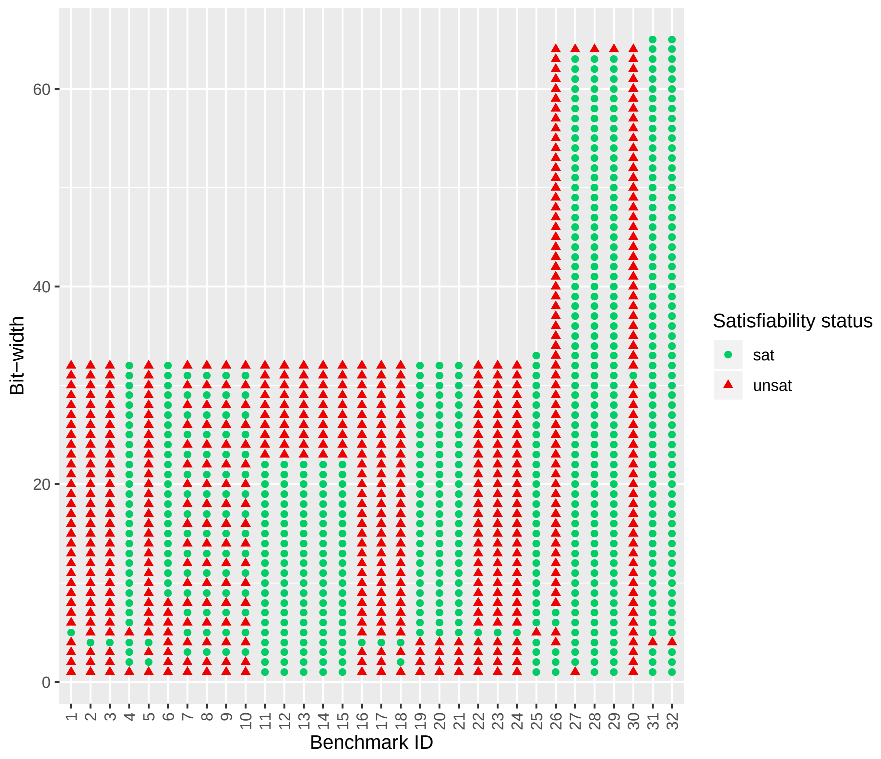


Figure 10.3: For each of the 32 benchmarks that have a different satisfiability result when reducing their bit-width to 4 bits or more, the plot shows satisfiability statuses for all their reduced versions.

32bit formulas change their satisfiability status after reducing their bit-width to 6 bits or more.

Figure 10.3 presents the satisfiability status of each reduction for all 32 formulas that changed the status after reducing the bit-width to 4 bits or more. Although the plot does not show names of the respective formulas due to the available space, the names can be found on the accompanying web page[1]. Note

1 http://fi.muni.cz/~xstrejc/lpar2018/

that most of these formulas are unsatisfiable; this is caused by the simple fact that most of the formulas in the whole benchmark set are unsatisfiable. The plot contains four outstanding groups of formulas:

- Formulas 7–10 (`intersection-example-onelane.proof-node{19355, 20770,46589,54847}` from *2017-Preiner-keymaera*). These formulas are unsatisfiable for even bit-widths and satisfiable for odd bit-widths because they contain subformulas equivalent to

  $$\begin{aligned}&\left(c^{[n]} = (x^{[n]} \cdot x^{[n]})/_s(2^{[n]} \cdot y^{[n]})\right) \ \land\\ &\left(0^{[n]} \leq_s y^{[n]}\right) \ \land\\ &\left(y^{[n]} >_s y^{[n]} + c^{[n]}\right) \ \land\\ &\left(y^{[n]} \leq_s c^{[n]}\right).\end{aligned}$$

  For $n \geq 2$, this subformula entails the formula

  $$c^{[n]} \geq_s (2^{n-2})^{[n]}$$

  and thus also a formula

  $$\left((x^{[n]} \cdot x^{[n]})/_s(2^{[n]} \cdot y^{[n]})\right) \ \geq_s \ (2^{n-2})^{[n]},$$

  which is unsatisfiable for even bit-widths, but satisfiable for odd bit-widths by setting $x^{[n]} \mapsto bv_n(2^{(n-1)/2})$ and $y^{[n]} \mapsto bv_n(2^{(n-1)} - 1)$.

- Formulas 11–15 (`jain_7_true_unreach-call_true-no-overflow_i_{ 215,242,245,262,475}` from *heizmann-ua*). Satisfiability of these formulas differs for bit-widths less than 23 and at least 23, because these formulas contain the numeral (presented here in binary)

  $$1111\,1111\,1100\,0000\,0000\,0000\,0000\,0000,$$

  which gets reduced to 0 after reduction to less than 23 bits.

- Formulas 27–29 (`check_eq_bvashr0_64bit` and `check_ne_{bvlshr0, bvshl0}_64bit` from *2018-Preiner-cav18*). Satisfiability of these formulas is different for the original bit-width of 64 bits and for almost all smaller bit-widths because the formulas contain a subformula similar to $(x^{[64]} <_u 64^{[64]}) \rightarrow \psi$, where ψ contains a subterm of the form $t \ll x^{[64]}$.

- Formula 30 (`mmedia_gsm610_gsm6102.c` from *wintersteiger*). Satisfiability of this formula is different for reduction to 31 bits, as it contains a subformula of the form

  $$x^{[32]} \leq_s 0100\,0000\,0000\,0000\,0000\,0000\,0000\,0000,$$

  in which the second argument has a negative sign-bit precisely for the reduction to 31 bits.

Detailed results together with the raw data files and scripts we used to produce them can be found at:

`http://fi.muni.cz/~xstrejc/lpar2018/`.

10.3 DISCUSSION

The experimental evaluation in the previous section shows that the satisfiability of the vast majority of quantified bit-vector formulas remains the same even after reducing their maximal bit-widths to a very small number of bits. In our opinion, this observation can be helpful in several ways:

- Because satisfiability of some formulas can be decided even without using the original bit-width, more fine-grained computational complexity of deciding their satisfiability could be identified. Currently, the known results of computational complexity are in term of the size of the input formula, from which the bit-widths are inseparable. In contrast, *parameterized computational complexity* [DF99] could be examined to show how the complexity depends on various parameters such as the bit-width, the size of the largest constant, the number of used function symbols, number of quantifier alternations, etc.

- As a practical use of the previous point, it could be possible for some formulas to compute a bit-width for which the reduced formula is equisatisfiable to the input one. Such a bit-width could be used to decide the satisfiability without using the original bit-width.

- Similarly to the case of the approximation framework of Zeljić et al. for quantifier-free formulas, the performance of SMT solvers for quantified bit-vectors could be improved by first solving a reduced version of the input formula and then checking the result against the original formula. For example, the solver Boolector computes symbolically represented Skolem functions, which certify satisfiability, and Herbrand functions, which certify unsatisfiability [PNB17]. These functions can be computed from a reduced formula and their validity can be checked against the original formula. More generally, in an SMT solver based on quantifier instantiation such as Boolector, CVC4 [Nie+18b], or Z3 [WHM13], the set of quantifier instances that are sufficient to decide satisfiability of the reduced formula can be checked against the original formula.

  Preliminary investigation of this line of research is presented in the next chapter.

11 SPEEDING UP QUANTIFIED BIT-VECTOR SMT SOLVERS BY BIT-WIDTH REDUCTIONS AND EXTENSIONS

In the previous chapter, we have experimentally confirmed that satisfiability of quantified bit-vector formulas is usually stable under bit-width reduction. Moreover, formulas with reduced bit-widths can often be solved faster. Building on these observations, this chapter presents a new technique for solving satisfiability of quantified bit-vector formulas, which uses formulas with the reduced bit-widths.

Intuitively, the technique consists of the following steps:

1. Reduce bit-widths of all variables and numerals in the input formula to some small bit-width.

2. Decide satisfiability of the reduced formula using a standard decision procedure. If the reduced formula is satisfiable, obtain its model, which assigns terms to all existentially quantified variables. If it is unsatisfiable, obtain its countermodel, i.e., an assignment of terms to all universally quantified variables.

3. Extend the model (or the countermodel) to the original bit-widths.

4. Check whether the extended (counter)model is also a (counter)model of the original formula. If the extended model is a model of the original formula, then the formula is satisfiable. If the extended countermodel is a countermodel of the original formula, then it is unsatisfiable. In the remaining cases, repeat the process with increased bit-widths in the reduced formula.

The technique has some similarities with the approximation framework of Zeljić et al. [ZWR17; Zel+18], which also reduces the precision of a given formula, computes a model of the reduced formula, and checks if it is a model of the original formula. However, the framework considers only quantifier-free formulas and hence the obtained models are just elements of the considered domains. In comparison, the models in our settings also provide interpretations of all Skolem function symbols, which correspond to the values of the existential variables of the original formula. Furthermore, the approximation framework of Zeljić et al. does not work with countermodels but processes unsatisfiable cores of reduced formulas instead.

As in the technique of Zeljić et al., the reduced formulas may not entail the original formula nor be entailed by it. This technique thus produces *mixed approximations* of the original formula, and not its underapproximations or overapproximations. This difference sets the technique apart from the technique using underapproximations and overapproximations, which was discussed in Chapter 5. On the one hand, the mixed approximations are beneficial because bit-widths of *all* variables are reduced, and thus the technique guarantees small upper bounds on the number of bits in the formula. On the other hand, neither satisfiability nor unsatisfiablity of a mixed approximation directly proves the satisfiability or unsatisfiability of the input formula. This requires checking of models or countermodels.

The structure of the chapter is as follows. Section 11.1 explains the necessary notions of symbolic model and countermodel. The detailed description of (counter)model extension is given in Section 11.2. The algorithm is precisely formulated in Section 11.3. Section 11.4 presents our proof-of-the-concept implementation and it discusses its practical aspects: e.g., how to get a countermodel and what to do with an incomplete model. Experimental evaluation of the technique can be found in Section 11.5. It shows that the presented technique can improve performance of the considered state-of-the-art solvers for quantified bit-vector formulas, namely Boolector [Nie+18a], CVC4 [Bar+11], and Q3B, on various families of both satisfiable and unsatisfiable quantified bit-vector formulas from the SMT-LIB repository.

11.1 SYMBOLIC MODELS AND COUNTERMODELS

This section defines notions of *symbolic model* and *symbolic countermodel*, which are crucial for our approach.

For a satisfiable formula φ without uninterpreted functions, a model $\mathcal{M}$ of *skolemize*(φ) assigns

- to each free variable $y^{[m]}$ of φ a bit-vector $\mathcal{M}(y^{[m]})$ of bit-width m and
- to each Skolem function symbol $f_{x^{[n]}}$, which corresponds to an existentially quantified variable $x^{[n]}$ in the formula φ, a function $\mathcal{M}(f_{x^{[n]}})$ whose arguments correspond to all universally quantified variables before $x^{[n]}$.

However, the functions $\mathcal{M}(f_{x^{[n]}})$ may be arbitrary functions in the mathematical sense. To be able to work with the model and substitute its values $\mathcal{M}(f_{x^{[n]}})$ into a formula, we use the notion of a *symbolic model*, in which the function $\mathcal{M}(f_{x^{[n]}})$ is represented symbolically by a term. Namely, $\mathcal{M}(f_{x^{[n]}})$ is a bit-vector term of bit-width n, whose free variables may be only the universal variables that are quantified before $x^{[n]}$ in the original formula φ. In the further text, we identify the Skolem function symbols with the corresponding variables. Namely, we treat the symbolic models as if they assign a term not to the corresponding Skolem function $f_{x^{[n]}}$, but directly to the existentially quantified variable $x^{[n]}$. For example:

Example 11.1. *The formula*

$$\varphi = \forall x^{[32]} \exists y^{[32]} \, (x^{[32]} + y^{[32]} = 0^{[32]})$$

has a symbolic model $\{y^{[32]} \mapsto -y^{[32]}\}$.

For an unsatisfiable formula φ, the dual notion to the symbolic model is a *symbolic countermodel*. The symbolic countermodel of a formula φ is a symbolic model of the formula $\neg\varphi'$, where φ' is the result of adding all implicit existential quantifiers to φ. In other words, a symbolic countermodel is mapping that assigns to each *universally* quantified variable $x^{[n]}$ in φ a term of bit-width n whose free variables may be only the *free* variables of φ or *existentially* quantified variables that are quantified before $x^{[n]}$ in the original formula φ.

Example 11.2. *The formula*

$$\forall y^{[32]} \, (x^{[32]} + y^{[32]} = 0^{[32]})$$

has a symbolic countermodel $\{y^{[32]} \mapsto -x^{[32]} + 1^{[32]}\}$.

Because all elements of symbolic models and countermodels are terms, we can substitute them into a given formula. We define this notion more generally to allow substitution of an arbitrary assignment that assigns terms to variables of the formula. For each such assignment $\mathcal{A}$ and a formula φ, we denote as $\mathcal{A}(\varphi)$ the result of simultaneous substitution of the term $\mathcal{A}(x^{[n]})$ for each variable $x^{[n]}$ from the domain of $\mathcal{A}$ and removing all quantifiers for the substituted variables. For example, given $\mathcal{A} = \{y^{[32]} \mapsto -x^{[32]}\}$, the value of

$$\mathcal{A}(\forall x^{[32]} \exists y^{[32]} \, (x^{[32]} + y^{[32]} = 0^{[32]}))$$

is $\forall x^{[32]} \, (x^{[32]} + -x^{[32]} = 0^{[32]})$.

11.2 EXTENDING BIT-WIDTH OF A SYMBOLIC MODEL

If a reduced formula is satisfiable and its symbolic model $\mathcal{M}$ is obtained, it cannot be directly substituted into the original formula. It first needs to be *extended* to the original bit-widths. Namely, some bit-widths need to be increased to match those of the original formula φ. Intuitively, for each result $\mathcal{M}(x) = t$, where the original bit-width of the variable x is n, we

1. increase bit-widths of all variables in t to match the bit-widths in the original formula φ,

2. for each operation whose arguments need to have the same bit-width, such as $+$ or $\times$, we increase bit-width of the argument with the smaller bit-width to match the bit-width of the other argument,

3. change the bit-width of the resulting term to match the bit-width of the original variable $x^{[n]}$.

In the formalization, we need to know bit-widths of the variables of the original formula. Therefore, for a formula φ, we introduce the function bws_φ that to each variable name x used in φ assigns its bit-width in φ. For example, $\mathrm{bws}_{x^{[32]}+y^{[32]}=0^{[32]}}(x) = 32$. To ensure that the function bws_φ is well-defined, we suppose that the input formula φ does not contain any variable with two different bit-widths.

We also use the function *changeBW*, which increases or decreases the bit-width of the given term t to the given bit-width. I.e.,

The operation `zero_extend` *can also be used instead of* `sign_extend`.

$$changeBW(t,n) = \begin{cases} t, & \text{if } \mathrm{bw}(t) = n, \\ \mathtt{sign\_extend}_{n-\mathrm{bw}(t)}(t), & \text{if } \mathrm{bw}(t) < n, \\ \mathtt{extract}_{(n-1),0}(t), & \text{if } \mathrm{bw}(t) > n. \end{cases}$$

We now describe how to for each element t of the reduced model compute a term $\overline{t}$, which uses only the variables of the original formula and is well-sorted, i.e., how to implement steps 1. and 2. of the extension described above. The computation of $\overline{t}$ proceeds by recursion on the structure of the term.

Bit-widths of numerals will be increased later if some operation requires it.

As the base cases, we keep the bit-width of all numerals and extend the bit-width of all variables to their original bit-widths, which are used in φ:

$$\overline{c^{[n]}} = c^{[n]},$$
$$\overline{x^{[n]}} = x^{[bws_\varphi(x)]}.$$

For the operations whose arguments are not required to have the same bit-widths, we proceed by the straightforward homomorphic extension:

$$\begin{aligned} \overline{op(t_1)} &= op(\overline{t_1}) \text{ for } op \in \{-, \sim\}, \\ \overline{\mathtt{concat}(t_1, t_2)} &= \mathtt{concat}(\overline{t_1}, \overline{t_2}), \\ \overline{\mathtt{extract}_{j,i}(t_1)} &= \mathtt{extract}_{j,i}(\overline{t_1}), \\ \overline{ext_n(t_1)} &= ext_n(\overline{t_1}) \text{ for } ext \in \{\mathtt{zeroExtend}, \mathtt{signExtend}\}. \end{aligned}$$

For the operations whose arguments are required to have the same bit-widths, we may need to extend the shorter of these arguments:

$$\begin{aligned} \overline{t_1 \diamond t_2} = {} & changeBW\left(\overline{t_1}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right) \diamond \\ & changeBW\left(\overline{t_2}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right) \\ & \text{for } \diamond \in \{\&, |, +, \times, /_u, /_s, \%_u, \%_s, \ll, \gg_u, \gg_s\}, \\ \overline{\mathtt{ite}(\varphi, t_1, t_2)} = {} & \mathtt{ite}\Big(\overline{\varphi}, \\ & \quad changeBW\left(\overline{t_1}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right), \\ & \quad changeBW\left(\overline{t_2}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right)\Big). \end{aligned}$$

In the previous case, we also have to extend bit-widths in the formula φ, which is the first argument of the if-then-else function. We suppose that this formula is quantifier-free, which is always the case for the existing SMT solvers. We therefore also define the extension for formulas, i.e., terms of sort *Bool*. This uses precisely the same approach as above:

$$\begin{aligned}
\overline{\top} &= \top, \\
\overline{\bot} &= \bot, \\
\overline{t_1 \bowtie t_2} &= \mathit{changeBW}\left(\overline{t_1}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right) \bowtie \\
&\quad \mathit{changeBW}\left(\overline{t_2}, \max\left(\mathrm{bw}(\overline{t_1}), \mathrm{bw}(\overline{t_2})\right)\right) \\
&\quad \text{for } \bowtie \in \{=, \leq_u, <_u, \leq_s, <_s\}, \\
\overline{\neg\varphi_1} &= \neg\overline{\varphi_1}, \\
\overline{\varphi_1 \diamond \varphi_2} &= \overline{\varphi_1} \diamond \overline{\varphi_2} \text{ for } \diamond \in \{\wedge, \vee\},
\end{aligned}$$

The defined extension function can be used to extend all terms in the given symbolic model of the reduced formula. In particular, we define the model extension $\mathit{extendM}(\mathcal{M})$ for each variable x in the domain of $\mathcal{M}$ by

$$\mathit{extendM}(\mathcal{M})(x) = \mathit{changeBW}(\overline{\mathcal{M}(x)}, \mathit{bws}_{\varphi}(x)).$$

Example 11.3. *Consider a formula φ that contains variables $x^{[8]}$, $y^{[8]}$, and $z^{[4]}$. Suppose that we have the following model $\mathcal{M}$ of $\mathit{reduceF}(\varphi, 4)$:*

$$\begin{aligned}
\mathcal{M} = \{&x^{[4]} \mapsto y^{[4]} + 3^{[4]}, \\
&y^{[4]} \mapsto z^{[4]}, \\
&z^{[4]} \mapsto y^{[4]}\}.
\end{aligned}$$

Then the candidate extended model $\mathit{extendM}(\mathcal{M})$ is

$$\begin{aligned}
\mathit{extendM}(\mathcal{M}) = \{&x^{[8]} \mapsto y^{[8]} + 3^{[8]}, \\
&y^{[8]} \mapsto \mathtt{sign\_extend}_4(z^{[4]}), \\
&z^{[4]} \mapsto \mathtt{extract}_{0,3}(y^{[8]})\}.
\end{aligned}$$

11.3 ALGORITHM

In this section, we propose an algorithm that uses formulas with reduced bit-widths to decide satisfiability of an input formula. In the first subsection, we describe a simpler approach that can only decide that a formula is satisfiable. The following subsection dualizes this approach to unsatisfiable formulas. We then show how to combine these two approaches in a single algorithm, which is able to decide both satisfiability and unsatisfiability of a formula.

11.3.1 *Checking Satisfiability Using Reductions and Extensions*

Having defined the functions *reduceF* (see Section 10.1), which reduces bit-widths in a formula, and *extendM*, which extends bit-widths in a model of the reduced formula, it is fairly straightforward to formulate an algorithm that can decide satisfiability of a formula using reduced bit-widths.

This algorithm first reduces the bit-widths in the input formula φ, thus obtains a reduced formula φ_{red}, and checks its satisfiability. If the formula is not satisfiable, the algorithm computes a new reduced formula φ_{red} with an increased bit-width and repeats the process. If, on the other hand, the reduced formula φ_{red} is satisfiable, the algorithm obtains its symbolic model $\mathcal{M}$, which assigns a term to each existentially quantified and free variable of the formula φ_{red}. The model is then extended to the original bit-widths of the variables in the formula φ and the extended model is substituted into the original formula φ, yielding a formula φ_{subst}. The formula φ_{subst} may not be quantifier-free, but it contains only universally quantified variables and no free variables. The formula φ_{subst} may therefore be checked for satisfiability by a solver for quantifier-free bit-vectors: the solver can be called on the formula that results from removing all quantifiers from the formula $\neg\varphi_{subst}$. Since the formula φ_{subst} is closed, the satisfiability of $\neg\varphi_{subst}$ implies unsatisfiability of φ_{subst} and vice-versa. Finally, if the formula φ_{subst} is satisfiable, so is the original formula. If the formula φ_{subst} is not satisfiable, the process is repeated with an increased bit-width.

This call may actually use a different SMT solver than the one used for checking satisfiability of φ_{red}.

Example 11.4. *Consider the formula $\varphi \equiv \forall x^{[32]} \exists y^{[32]} (x^{[32]} + y^{[32]} = 0^{[32]})$. Reduction to 2 bits yields the formula reduceF$(\varphi, 2) \equiv \forall x^{[2]} \exists y^{[2]} (x^{[2]} + y^{[2]} = 0^{[2]})$. An SMT solver can decide that this formula is satisfiable and its symbolic model is $\{y^{[2]} \mapsto -x^{[2]}\}$. An extended candidate model is then $\{y^{[32]} \mapsto -x^{[32]}\}$. After substituting this candidate model into the formula, one gets the formula $\varphi_{subst} \equiv \forall x^{[32]} (x^{[32]} + (-x^{[32]}) = 0^{[32]})$. Negating the formula φ_{subst} and removing all the quantifiers yields the quantifier-free formula $(x^{[32]} + (-x^{[32]}) \neq 0^{[32]})$, which can be decided unsatisfiable by an SMT solver. Therefore, the formula φ_{subst} is satisfiable and, in turn, the original formula φ was satisfiable.*

Note that in this case, satisfiability of φ_{subst} can be decided purely by simple rewriting rules.

The correctness of the approach is guaranteed by the following theorem.

Theorem 11.1. *Let φ be a formula in the negation normal form and $\mathcal{M}$ a mapping that assigns terms only to free and existentially quantified variables of φ. Then satisfiability of $\mathcal{M}(\varphi)$ implies satisfability of φ.*

Proof. In the formula $\mathcal{M}(\varphi)$, some subformulas of form $\exists x\, (\psi)$ are replaced by formulas $\psi[x \leftarrow \mathcal{M}(x)]$ and some free variables are replaced by given terms. The claim follows from the following three observations

- The entailment $\psi[x \leftarrow t] \models \exists x\, (\psi)$ holds for an arbitrary formula ψ and a term t of the same sort as the variable x.
- If a formula after replacing some of its free variables by some terms is satisfiable, the original formula must have been satisfiable.
- All the logical operations $\wedge, \vee, \forall, \exists$ are monotnic. I.e., for all $\psi_1 \models \psi_1'$ and $\psi_2 \models \psi_2'$:

$$\begin{aligned}
&\psi_1 \wedge \psi_2 \models \psi_1' \wedge \psi_2',\\
&\psi_1 \vee \psi_2 \models \psi_1' \vee \psi_2',\\
&\forall x\, (\psi_1) \models \forall x\, (\psi_1'),\\
&\exists x\, (\psi_1) \models \exists x\, (\psi_1').
\end{aligned}$$

Although the negation is not monotonic, this is not a problem since the formula φ is in the negation normal form. Therefore, no negation in φ is applied to a formula that contains a subformula of form $\exists x\,(\psi)$. □

11.3.2 *Dual Solver*

The described algorithm can improve performance only for satisfiable formulas since it can only decide a formula as unsatisfiable after computing its satisfiability for the original bit-width. However, similarly to the dual solver employed in Boolector, a dual version of the described algorithm can be used to improve performance on unsatisfiable formulas. In the dual algorithm, one can decide unsatisfiability of a formula by computing a countermodel from the reduced formula and verifying it against the original formula. In particular, if the solver decides that the reduced formula φ_{red} is unsatisfiable and $\mathcal{C}$ is its countermodel, one can again extend the countermodel $\mathcal{C}$, substitute the extended countermodel into the original formula, obtaining a formula φ_{subst}, which contains only existentially quantified variables. If the formula φ_{subst} is unsatisfiable, the original formula φ must have been unsatisfiable. If the formula φ_{subst} is satisfiable, the process is repeated with an increased bit-width.

Example 11.5. *Consider the formula* $\varphi = \forall y^{[32]}\,(x^{[32]} + y^{[32]} = 0^{[32]})$. *Reduction to one bit yields the formula* $reduceF(\varphi, 1) = \forall y^{[1]}\,(x^{[1]} + y^{[1]} = 0^{[1]})$. *This formula can be decided as unsatisfiable by an SMT solver and its countermodel is* $\{y^{[1]} \mapsto -x^{[1]} + 1^{[1]}\}$. *The extension of this countermodel to the original bit-widths is then* $\{y^{[32]} \mapsto -x^{[32]} + 1^{[32]}\}$. *After substituting this candidate countermodel to the original formula, one obtains the quantifier-free formula* $\varphi_{subst} = (x^{[32]} + (-x^{[32]} + 1^{[32]}) = 0^{[32]})$, *which is unsatisfiable. The original formula* φ *is thus unsatisfiable.*

Similarly to the claim in the previous section, correctness of the dual solver is guaranteed by the following theorem.

Theorem 11.2. *Let* φ *be a formula in the negation normal form and* $\mathcal{M}$ *a mapping that assigns terms to some universally quantified variables of* φ. *Then unsatisfiability of* $\mathcal{M}(\varphi)$ *implies unsatisfiability of* φ.

Proof. The proof is dual to the proof of Theorem 11.1. □

11.3.3 *Combined Solver*

We now show how to combine the two previously mentioned approaches into one algorithm. In the rest of this section, we suppose that there exists an SMT solver that can return symbolic models for satisfiable quantified bit-vector formulas and countermodels for unsatisfiable ones. On the one hand, this assumption significantly improves the presentation and simplicity of the proposed algorithm. On the other hand, there currently is no such solver. However, we show in Section 11.4 how this assumption can be weakened to only a solver that can return symbolic models for satisfiable formulas.

```
checkUsingReductions($\varphi$)
{
  bw ← 1
  while (bw <= maxBW($\varphi$))
  {
    $\varphi_{red} \leftarrow reduceF(\varphi, \text{bw})$

    (result, assignment) ← solve($\varphi_{red}$)
    $\mathcal{A} \leftarrow extendM(\text{assignment})$
    $\varphi_{subst} \leftarrow \mathcal{A}(\varphi)$

    if (result == sat) {
      //primal solver
      $\varphi_{subst}$ ← removeQuantifiers($\neg\varphi_{subst}$)
      verificationResult ← verify($\varphi_{subst}$)
      if (verificationResult == UNSAT) return SAT
    } else if (result == unsat) {
      //dual solver
      verificationResult ← verify($\varphi_{subst}$)
      if (verificationResult == UNSAT) return UNSAT
    }

    bw ← min(2*bw, maxBW($\varphi$))
  }
}
```

Listing 11.1: Pseudocode of the combined solver.

We call the described solver as a *model-generating solver*. Let us denote as solve its function, which returns solve(φ) = (sat, *model*) for each satisfiable formula φ and solve(φ) = (unsat, *countermodel*) for each unsatisfiable one. The proposed approach also uses SMT queries to test the satisfiability of $\neg\varphi_{subst}$. Generally, this query can be checked by a different SMT solver than the model-generating one; we call the second solver *model-validating solver* and denote its function, which for a given formula returns either sat or unsat, as verify.

Using these two solvers, the algorithm presented in Listing 11.1 combines the techniques of the two preceding sections. This algorithm first checks satisfiability of the reduced formula and according to the result tries to validate either its symbolic model or symbolic countermodel.

11.4 IMPLEMENTATION

Based on the described algorithm for the combined solver, we have implemented a proof-of-the-concept experimental tool, which solves quantified bit-vector formulas using bit-width reductions and extensions. However, our implementation differs in several aspects from the described algorithm. This section explains all these differences and provides more details about the implementation.

11.4.1 *Model-Generating Solver*

As the model-generating SMT solver, we have used Boolector 3.0.0 as it can return symbolically expressed Skolem functions as models of satisfiable quantified formulas, which is crucial for our approach. To the best of our knowledge, no other SMT solver for quantified bit-vectors offers this functionality. However, the current version of Boolector does not satisfy all the other requirements that we imposed on the model-generating solver.

First, the symbolic model $\mathcal{M}$ returned by Boolector may not contain terms for all existentially quantified variables in the input formula φ. Therefore, the formula $\varphi_{subst} = \mathcal{M}(\varphi)$ may still contain existentially quantified variables. This prohibits the explained check of its satisfiability using a solver for quantifier-free formulas. Our implementation can solve this problem in two different ways:

- we check satisfiability of the formula φ_{subst} by a model-validating solver that supports *quantified* bit-vector formulas; or
- we substitute the bit-vector term $0^{[n]}$ for each existentially quantified variable $x^{[n]}$ that remains in the formula φ_{subst}. This allows using a solver for quantifier-free bit-vectors as the model-validating solver.

Second, Boolector returns symbolic models only for satisfiable formulas and cannot return symbolic countermodels. We alleviate this problem by running two parallel instances of Boolector: one on the original formula φ and one on the formula $\neg\varphi'$, where φ' is the result of existentially quantifying all free variables in φ. We then use only the result of the solver that decides that the formula is satisfiable; if φ is satisfiable, we get its symbolic model, if $\neg\varphi'$ is satisfiable, we get its symbolic model, which is a symbolic countermodel of φ. Effectively, this is equivalent to running the proposed algorithm without the dual solver on φ and $\neg\varphi'$ in parallel. This is similar to the dual solver employed internally by Boolector. However, we treat Boolector as a black-box and use only its public interface.

11.4.2 *Portfolio Solver*

Additionally, to ensure that the performance of the original model-validating solver is not degraded by computing reductions, we propose to run in parallel also the original model-validating solver besides the other two solvers that use bit-width reductions. As is standard for portfolio solvers, the result of the first of the three solvers that decides the satisfiablity of the formulas is returned. The schematic overview of the portfolio solver is presented in Figure 11.1.

Thanks to the fact that the original solver is one of the three parallel solvers, the wall time of the portfolio solver can be worsened only by the time needed to initialize the parallel threads, which is constant. On the other hand, the CPU time of the portfolio solver can be even three times worse than the CPU time of the original solver.

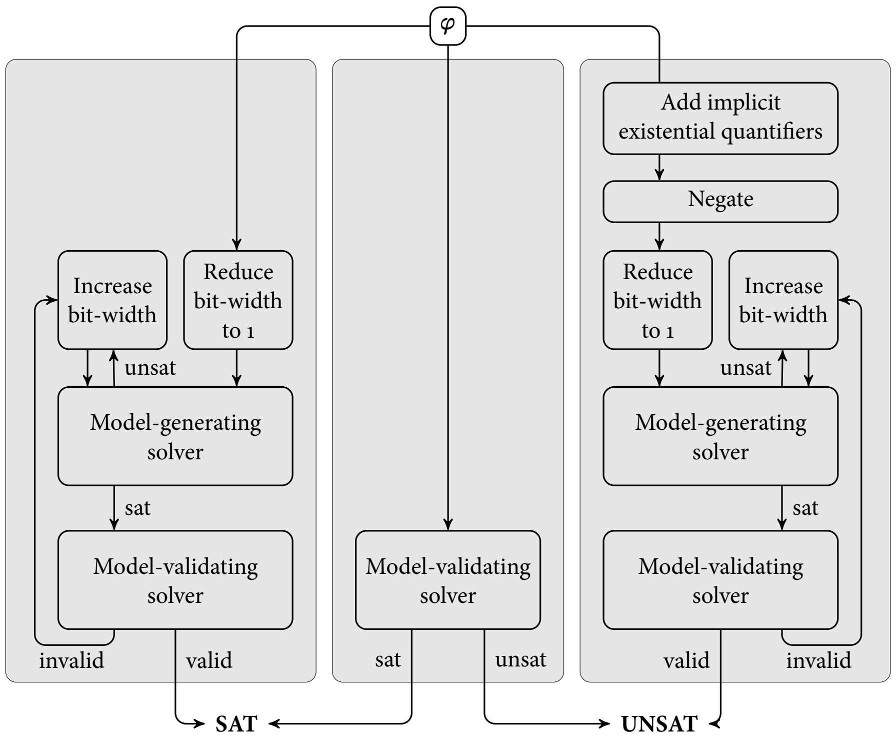


Figure 11.1: High-level overview of the portfolio solver. The three shaded areas are executed in parallel and the first result is returned.

We have implemented the described portfolio solver partly in C++ and partly in Python. C++ was used to implement the solvers that use reduced formulas and are executed in parallel with the model-validating solver. The implementation is experimental and uses C++ API of the SMT solver Z3 to parse the input formula in the SMT-LIB format. The Z3 API is also used in the implementation of formula reductions. The extension of the model and substitution of the extended model to the original formula is achieved by a simple, inefficient, and incomplete text manipulations with the SMT-LIB text format. We have further used a simple Python wrapper to implement the tool that runs three parallel threads and collects their results. The complete tool is available from

`https://gitlab.fi.muni.cz/xjonas/BWReducingSolver`.

Note that the current implementation of the portfolio solver is experimental and has many limitations. Namely, the current version supports only formulas in which all variables have the same bit-width. On the other formulas, only the model-validating solver is used.

11.5 EXPERIMENTAL EVALUATION

We have tested the performance of the implemented portfolio solver with three model-validating solvers: Boolector 3.0.0 [Nie+18a], CVC4 1.6 [Bar+11], and

Q3B 1.0. Recall that the model-generating solver is the same in all of the cases, namely Boolector 3.0.0. For the evaluation, we have again used all 5751 quantified bit-vector formulas from the SMT-LIB benchmark repository [BFT16]. As described in previous chapters, this repository consists of several benchmark families. All the presented results in this evaluation are divided according to these benchmark families.

All experiments were again performed on a Debian machine with two six-core Intel Xeon E5-2620 2.00GHz processors and 128 GB of RAM. Each benchmark run was limited to use 16 GB of RAM and 5 minutes of *wall* time. All measured times are wall times. For reliable benchmarking we employed the tool BenchExec [BLW15].

11.5.1 *Boolector*

First, we have evaluated the effect of reductions on the performance of Boolector 3.0.0. as the model-validating solver. We have compared the following three solvers:

- `btor`: the vanilla Boolector,
- `btor-r`: the portfolio solver using Boolector as the model-validating solver, without substituting zeroes for values not present in the model, as described in Subection 11.4.1,
- `btor-rz`: the portfolio solver using Boolector as the model-validating solver, with substituting zeroes for values not present in the model.

Table 11.1 shows the numbers of solved formulas in the individual benchmark families for all three of these solvers. The portfolio solver using the bit-width reductions was able to solve 24 more formulas than Boolector itself. Note that this amounts to 8.4 % of the benchmarks unsolved by Boolector. After enabling the substitution of zeroes for unknown values in the model, the portfolio solver `btor-rz` was able to decide 6 less benchmarks than `btor-r`; however, it solved 3 benchmarks that could not be solved by `btor-r`.

In addition to solving more formulas, the proposed reductions also help to solve several formulas faster. The scatter plot in Figure 11.2 shows the comparison of wall times of `btor` and `btor-r` on the logarithmic scale. It can be seen that the proposed reductions significantly improve performance of Boolector on a non-trivial number of benchmarks. Furthermore, these improvements occur in multiple benchmarks families and in both satisfiable and unsatisfiable benchmarks alike.

Further, we have also investigated the reduced bit-width that was sufficient to improve the performance of Boolector. Among all executions of the portfolio solver `btor-r`, in total 558 benchmarks were decided by one of two parallel solvers that perform bit-width reductions faster than by the thread using only Boolector. From these 558 benchmarks, 122 were decided by using the bit-width of 1 bit; 286 using 2 bits; 97 using 4 bits; 24 using 8 bits; 4 using 16 bits; 24 using 32 bits; and 1 using 64 bits.

Family	Total	btor	btor-r	btor-rz
2017-Preiner-keymaera	4035	4017	4023	**4024**
2017-Preiner-psyco	194	**191**	**191**	**191**
2017-Preiner-scholl-smt08	374	296	**303**	301
2017-Preiner-tptp	73	69	**72**	69
2017-Preiner-ua	153	151	**152**	**152**
2018-Preiner-cav18	600	548	**554**	553
heizmann-ua	131	30	**31**	30
wintersteiger	191	**161**	**161**	**161**
Total	5751	5463	**5487**	5481

Table 11.1: Comparison of benchmarks solved by `btor`, `btor-r`, and `btor-rz` within the given timeout.

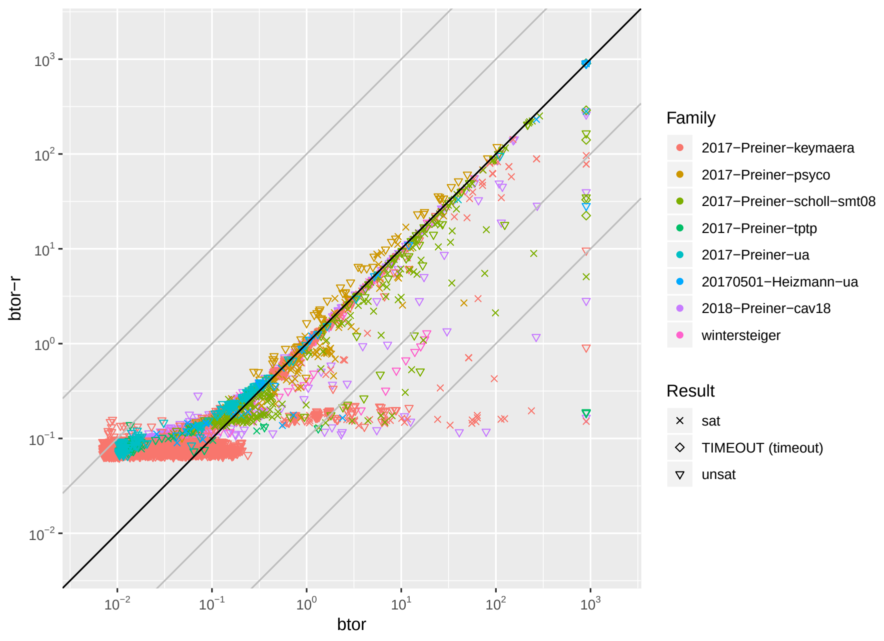


Figure 11.2: Scatter plot of wall times of the solver `btor` and the solver `btor-r`. Each point represents one benchmark, its color shows the benchmark family, and its shape shows its satisfiability. The gray lines represent hundredfold and thousandfold difference.

11.5.2 *CVC4 and Q3B*

We have also performed evaluations with CVC4 and Q3B as model-validating solvers. This yields the following four solvers:

- `cvc4`, `q3b`: the vanilla CVC4 and Q3B, respectively,
- `cvc4-r`, `q3b-r`: the portfolio solvers using CVC4 and Q3B, respectively, as the model-validating solver, without substituting zeroes for values not present in the model.

We did not evaluate the variant that substitutes zeroes, because its performance was inferior in the case of Boolector.

The comparison in this case, in which a model-generating solver differs from the model-validating solver, is more involved. For example, the direct comparison of `cvc4` and `cvc4-r` would be unfair and could be biased towards `cvc4-r`. The reason for this is that a benchmark can be decided within `cvc4-r` purely by Boolector itself as the model-generating solver. This happens when the formula is reduced to its original bit-width. In this case, the model containing terms for all of the variables of the original formula may be provided by Boolector and the substituted formula may become trivial to solve.

To eliminate this bias, we have not compared `cvc4` against `cvc4-r`, but the virtual-best solver from `btor` and `cvc4`, denoted as `btor|cvc4`, against the virtual-best solver from `btor` and `cvc4-red`, denoted as `btor|cvc4-red`. We thus investigate only the effect of reductions and not the case when the model-generating solver solves the input formula. Similarly, we compare the virtual-best solver `btor|q3b` against the virtual-best solver `btor|q3b-red`.

Table 11.2 shows the number of benchmarks solved by the compared solvers. In particular, reductions helped the virtual-best solver `btor|cvc4-red` to solve 9 more benchmarks than the solver `btor|cvc4`. This amounts to 7.4 % of the benchmarks unsolved by `btor|cvc4`. On the other hand, the reductions do not help the virtual-best solver `btor|q3b-red` to solve any new benchmarks.

Similarly to the case of Boolector, reductions also help `btor|cvc4-red` to decide several benchmarks faster than the solver `btor|cvc4` without reductions. This can be seen on the scatter plot in Figure 11.3. As the second scatter plot in Figure 11.4 shows, reductions also help Q3B to solve some benchmarks faster, although the effect is not as pronounced as with Boolector or CVC4.

11.5.3 *All Model-Validating Solvers*

Another interesting question is whether the reductions can improve the combination of Boolector, CVC4, and Q3B. To answer this, we have compared the virtual-best solver `btor|cvc4|q3b` and the virtual best solver `btor-r|cvc4-r|q3b-r`. The comparison of the numbers of solved formulas in the individual benchmark families can be found in Table 11.3. In total, reductions help to solve 4 more formulas, which could by solved neither by Boolector, CVC4, nor Q3B by itself without reductions. This amounts to 5.5 % of the unsolved formulas.

As in the previous sections, by using reductions, one can solve several formulas faster than by Boolector, CVC4, or Q3B without reductions. This can be seen in the scatter plot presented in Figure 11.5.

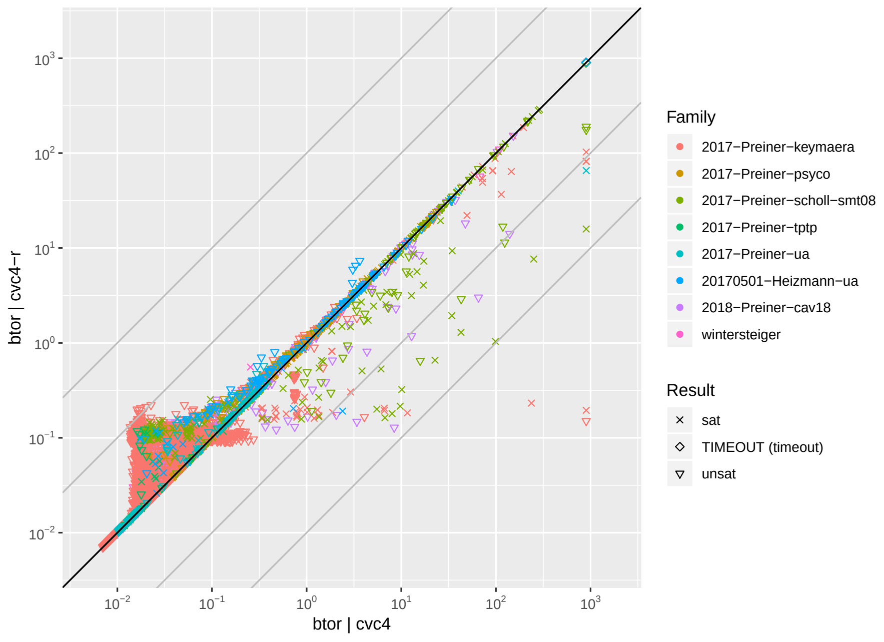


Figure 11.3: Scatter plot of wall times of the virtual-best solver `btor|cvc4` and the virtual-best solver `btor|cvc4-r`.

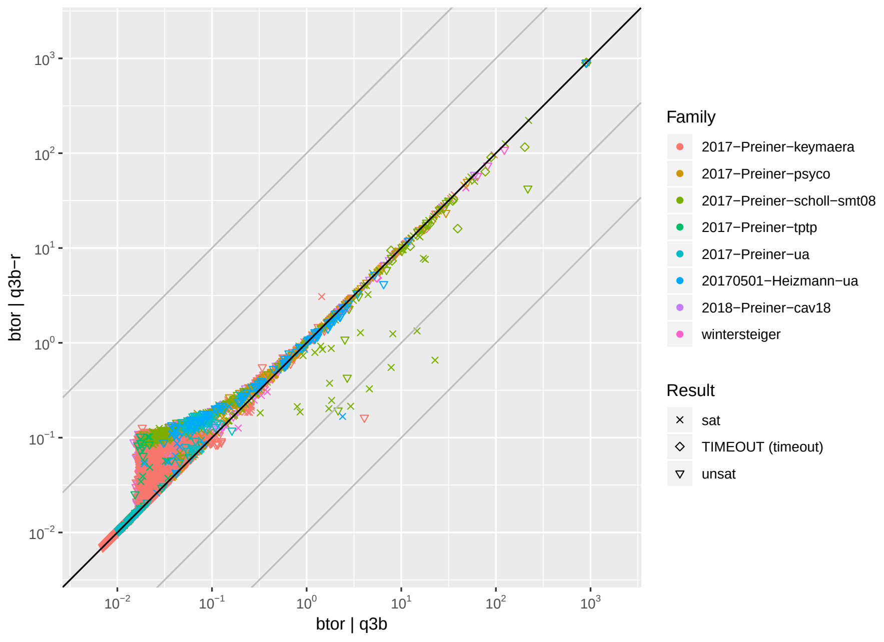


Figure 11.4: Scatter plot of wall times of the virtual-best solver `btor|q3b` and the virtual-best solver `btor|q3b-r`.

Family	Total	btor|cvc4	btor|cvc4-r	btor|q3b	btor|q3b-r
2017-Preiner-keymaera	4035	4023	4028	4025	4025
2017-Preiner-psyco	194	193	193	192	192
2017-Preiner-scholl-smt08	374	306	309	324	324
2017-Preiner-tptp	73	73	73	73	73
2017-Preiner-ua	153	152	153	153	153
20170501-Heizmann-ua	131	130	130	128	128
2018-Preiner-cav18	600	577	577	590	590
wintersteiger	191	176	176	182	182
Total	5751	5630	5639	5667	5667

Table 11.2: Comparison of benchmarks solved by `btor|cvc4`, `btor|cvc4-r`, `btor|q3b`, and `btor|q3b-r` within the given timeout.

Additional materials for all the experiments, including the experimental data, used scripts, and additional plots and tables are available from

`https://fi.muni.cz/~xjonas/PhdThesis/BWReductions`.

11.6 FUTURE WORK

We again stress out that the implementation that is used for evaluation of the approach is experimental. We plan to improve the implementation so that it supports also formulas that contain variables with different bit-widths. This requires reimplementation of the model extension, which is currently implemented by a simple text modifications. We thus plan to implement the function `extendT` exactly as it is described in Section 11.2 and perform new evaluation. This can only improve the results of the evaluation, which are already positive.

We also see several potential improvements of the described algorithm. For example, consider the formula

$$\varphi \;\equiv\; (x^{[32]} \times x^{[32]} = 0^{[32]}) \;\wedge\; (x^{[32]} \neq 0^{[32]}).$$

Observe that the formula $\mathit{reduceF}(\varphi, 2)$ has a model $\{x^{[2]} \mapsto bv_2(2)\}$, the formula $\mathit{reduceF}(\varphi, 4)$ has a model $\{x^{[4]} \mapsto bv_4(4)\}$, and the formula $\mathit{reduceF}(\varphi, 6)$ has a model $\{x^{[6]} \mapsto bv_6(8)\}$. The described approach does not solve this formula faster because none of these models can be extended to the model of the original formula by the function `extendM`. However, it is possible to observe that these models are related: a model of a formula $\mathit{reduceF}(\varphi, n+2)$ can be obtained from the model of the formula $\mathit{reduceF}(\varphi, n)$ by multiplying the

Family	Total	btor|cvc4|q3b	btor-r|cvc4-r|q3b-r
2017-Preiner-keymaera	4035	4029	4031
2017-Preiner-psyco	194	193	193
2017-Preiner-scholl-smt08	374	326	328
2017-Preiner-tptp	73	73	73
2017-Preiner-ua	153	153	153
20170501-Heizmann-ua	131	131	131
2018-Preiner-cav18	600	590	590
wintersteiger	191	183	183
Total	5751	5678	5682

Table 11.3: Comparison of benchmarks solved by `btor|cvc4|q3b` and `btor-r|cvc4-r|q3b-r` within the given timeout.

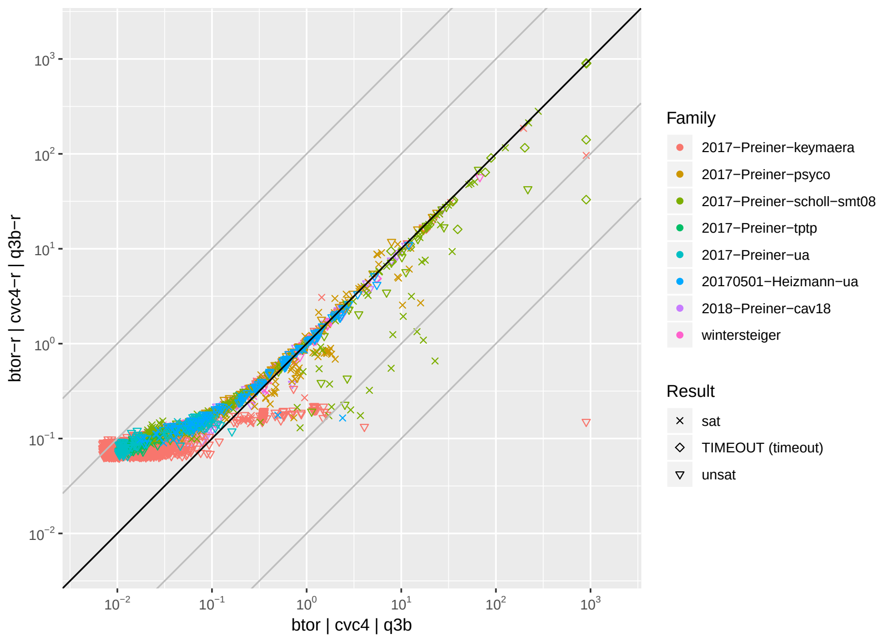


Figure 11.5: Scatter plot of wall times of the virtual-best solver `btor|cvc4|q3b` and the virtual-best solver `btor-r|cvc4-r|q3b-r`.

value of the variable x by 2, or in other words, shifting it by one bit to the left. This example motivates an improved version of the function `extendM`, which can also use the previously computed models of the formulas with the smaller bit-widths.

12 CONCLUSIONS

In this thesis, we have studied satisfiability of quantified formulas in the theory of fixed-size bit-vectors. The thesis has presented both theoretical and practical advances related to this problem.

We have identified the precise complexity class for the problem of solving satisfiability of quantified bit-vector formulas in which the bit-widths are encoded in binary or in decimal notation. Namely, this problem is polynomially equivalent to the satisfiability problem of second-order quantified Boolean formulas, which is known to be **AEXP**(poly)-complete. We have therefore answered the open question raised by Kovásznai et al. [KFB16].

Moreover, we have introduced an approach to solving satisfiability of quantified bit-vector formulas that is based on binary decision diagrams and approximations, in which some of the variables of the formula are represented by fewer bits. We have also extended this approach by abstractions in which for the results of selected bit-vector operations, only some of the bits are computed. Furthermore, we have extended known simplifications of formulas that leverage unconstrained variables in several aspects, one of which is extension to quantified formulas. We have implemented an SMT solver called Q3B, which incorporates all the mentioned techniques and we have experimentally shown that this SMT solver outperforms other state-of-the-art SMT solvers for quantified bit-vector formulas.

Finally, we have experimentally shown that satisfiability of the vast majority of quantified bit-vector formulas stays the same when the bit-widths of their variables are modified. We have also presented an approach that uses this observation to improve the performance of SMT solvers for quantified bit-vectors. The proposed approach first decides the satisfiability of the formula with reduced bit-widths and then verifies the result against the original formula. The preliminary evaluation of this approach shows that it can improve performance of the state-of-the-art SMT solvers.

[Alu+15] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-Guided Synthesis." In: *Dependable Software Systems Engineering*. 2015, pp. 1–25. DOI: 10.3233/978-1-61499-495-4-1 (cited on page 25).

[AS04] Gilles Audemard and Lakhdar Sais. "SAT Based BDD Solver for Quantified Boolean Formulas." In: *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*. 2004, pp. 82–89. DOI: 10.1109/ICTAI.2004.106 (cited on page 39).

[Bab+19] Junaid Babar, Chuan Jiang, Gianfranco Ciardo, and Andrew S. Miner. "Binary Decision Diagrams with Edge-Specified Reductions." In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. 2019, pp. 303–318. DOI: 10.1007/978-3-030-17465-1_17 (cited on page 92).

[Bar+09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Satisfiability*. IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825 (cited on page 7).

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016 (cited on pages 103, 123).

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017 (cited on pages 7, 29, 89, 106).

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14 (cited on pages 1, 114, 122).

[Bey19] Dirk Beyer. "Automatic Verification of C and Java Programs: SV-COMP 2019." In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. 2019, pp. 133–155. DOI: 10.1007/978-3-030-17502-3_9 (cited on page 5).

[BLW15] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Benchmarking and Resource Measurement." In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015, pp. 160–178. DOI: 10.1007/978-3-319-23404-5_12 (cited on pages 93, 107, 123).

[BW96] Beate Bollig and Ingo Wegener. "Improving the Variable Ordering of OBDDs Is NP-Complete." In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122 (cited on page 43).

[Bru10] Robert Brummayer. "Efficient SMT solving for bit vectors and the extensional theory of arrays." PhD thesis. Johannes Kepler University of Linz, 2010 (cited on page 69).

[BB09] Robert Brummayer and Armin Biere. "Effective Bit-Width and Under-Approximation." In: *Computer Aided Systems Theory - EUROCAST 2009, 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers*. 2009, pp. 304–311. DOI: 10.1007/978-3-642-04772-5_40 (cited on pages 45, 46).

[Bru08] Roberto Bruttomesso. "RTL Verification: From SAT to SMT(BV)." PhD thesis. University of Trento, 2008 (cited on pages 69, 72).

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819 (cited on pages 18, 19).

[Bry91] Randal E. Bryant. "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication." In: *IEEE Trans. Computers* 40.2 (1991), pp. 205–213. DOI: 10.1109/12.73590 (cited on page 44).

[Bry18] Randal E. Bryant. "Chain Reduction for Binary and Zero-Suppressed Decision Diagrams." In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. 2018, pp. 81–98. DOI: 10.1007/978-3-319-89960-2_5 (cited on page 92).

[Bry+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. "Deciding Bit-Vector Arithmetic with Abstraction." In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 2007, pp. 358–372. DOI: 10.1007/978-3-540-71209-1_28 (cited on pages 45, 104).

[Bur97] David M. Burton. *Elementary Number Theory*. International Series in Pure and Applied Mathematics. McGraw-Hill, 1997. ISBN: 9780070094635 (cited on page 72).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf (cited on page 1).

[Cad+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008), 10:1–10:38. DOI: 10.1145/1455518.1455522 (cited on page 1).

[Cha+16] Marek Chalupa, Martin Jonáš, Jiri Slaby, Jan Strejček, and Martina Vitovská. "Symbiotic 3: New Slicer and Error-Witness Generation - (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 946–949. DOI: 10.1007/978-3-662-49674-9_67 (cited on page 5).

[Cha+17] Marek Chalupa, Martina Vitovská, Martin Jonáš, Jiri Slaby, and Jan Strejček. "Symbiotic 4: Beyond Reachability - (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 2017, pp. 385–389. DOI: 10.1007/978-3-662-54580-5_28 (cited on page 5).

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. "Alternation." In: *J. ACM* 28.1 (1981), pp. 114–133. DOI: 10.1145/322234.322243 (cited on pages 29, 30, 32).

[Cim+13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 93–107. DOI: 10.1007/978-3-642-36742-7_7 (cited on page 1).

[Coo+13] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. "Ranking function synthesis for bit-vector relations." In: *Formal Methods in System Design* 43.1 (2013), pp. 93–120. DOI: 10.1007/s10703-013-0186-4 (cited on page 2).

[CFM12] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. "SMT-Based Bounded Model Checking for Embedded ANSI-C Software." In: *IEEE Trans. Software Eng.* 38.4 (2012), pp. 957–974. DOI: 10.1109/TSE.2011.59 (cited on page 1).

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. "A machine program for theorem-proving." In: *Commun. ACM* 5.7 (1962), pp. 394–397. DOI: 10.1145/368273.368557 (cited on page 72).

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: a theorem prover for program checking." In: *J. ACM* 52.3 (2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cited on page 24).

[DWM17] Tom van Dijk, Robert Wille, and Robert Meolic. "Tagged BDDs: Combining reduction rules from different decision diagram types." In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 2017, pp. 108–115. DOI: 10.23919/FMCAD.2017.8102248 (cited on page 92).

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999. ISBN: 978-1-4612-6798-0. DOI: 10.1007/978-1-4612-0515-9 (cited on page 111).

[Dut14] Bruno Dutertre. "Yices 2.2." In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49 (cited on page 1).

[Dut15] Bruno Dutertre. "Solving Exists/Forall Problems with Yices." In: *Workshop on satisfiability modulo theories*. 2015 (cited on pages 2, 23, 27).

[End01] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, 2001. ISBN: 978-0-12-238450-9 (cited on page 7).

[Fra10] Anders Franzén. "Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT." PhD thesis. University of Trento, 2010 (cited on pages 69, 71, 72).

[FKB13] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. "More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding." In: *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings*. 2013, pp. 378–390. DOI: 10.1007/978-3-642-38536-0_33 (cited on page 37).

[Frö+15] Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. "Stochastic Local Search for Satisfiability Modulo Theories." In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, pp. 1136–1143. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9896 (cited on page 104).

[Gad+18] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. "ESBMC 5.0: an industrial-strength C model checker." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018, pp. 888–891. DOI: 10.1145/3238147.3240481 (cited on page 1).

[GD07] Vijay Ganesh and David L. Dill. “A Decision Procedure for Bit-Vectors and Arrays.” In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, pp. 519–531. DOI: 10.1007/978-3-540-73368-3_52 (cited on page 1).

[GSV09] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. “Constraint-Based Invariant Inference over Predicate Abstraction.” In: *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*. 2009, pp. 120–135. DOI: 10.1007/978-3-540-93900-9_13 (cited on page 2).

[Had+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark Barrett, and Cesare Tinelli. “A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors.” In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 680–695. DOI: 10.1007/978-3-319-08867-9_45 (cited on page 23).

[Han+16] Miika Hannula, Juha Kontinen, Martin Lück, and Jonni Virtema. “On Quantified Propositional Logics and the Exponential Time Hierarchy.” In: *Proceedings of the Seventh International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2016, Catania, Italy, 14-16 September 2016*. 2016, pp. 198–212. DOI: 10.4204/EPTCS.226.14 (cited on page 34).

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. ISBN: 978-0-521-89957-4 (cited on pages 16, 41).

[Har16] Frederik Harwath. “A note on the size of prenex normal forms.” In: *Inf. Process. Lett.* 116.7 (2016), pp. 443–446. DOI: 10.1016/j.ipl.2016.03.005 (cited on page 16).

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Software Model Checking for People Who Love Automata.” In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 36–52. DOI: 10.1007/978-3-642-39799-8_2 (cited on page 93).

[Hut+07] Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. “Boosting Verification by Automatic Tuning of Decision Procedures.” In: *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*. 2007, pp. 27–34. DOI: 10.1109/FAMCAD.2007.9 (cited on page 1).

[Hyv+16] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. “OpenSMT2: An SMT Solver for Multi-core and Cloud Computing.” In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 547–553. DOI: 10.1007/978-3-319-40970-2_35 (cited on page 1).

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. “Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic.” In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 2009, pp. 668–674. DOI: 10.1007/978-3-642-02658-4_53 (cited on page 1).

[Joh01] Peer Johannsen. “Reducing bitvector satisfiability problems to scale down design sizes for RTL property checking.” In: *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, November 7-9, 2001*. 2001, pp. 123–128. DOI: 10.1109/HLDVT.2001.972818 (cited on page 104).

[Joh02] Peer Johannsen. “Speeding up hardware verification by automated data path scaling.” PhD thesis. University of Kiel, Germany, 2002 (cited on page 104).

[JS16] Martin Jonáš and Jan Strejček. “Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams.” In: *Theory and Applications of Satisfiability Testing – SAT 2016 – 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 267–283. DOI: 10.1007/978-3-319-40970-2_17 (cited on pages 2, 4, 5, 27, 100).

[JS17] Martin Jonáš and Jan Strejček. “On Simplification of Formulas with Unconstrained Variables and Quantifiers.” In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. 2017, pp. 364–379. DOI: 10.1007/978-3-319-66263-3_23 (cited on pages 4, 5, 76, 100).

[JS18a] Martin Jonáš and Jan Strejček. “Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers.” In: *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*. 2018, pp. 273–291. DOI: 10.1007/978-3-030-02508-3_15 (cited on pages 4, 5, 27, 100).

[JS18b] Martin Jonáš and Jan Strejček. “Is Satisfiability of Quantified Bit-Vector Formulas Stable Under Bit-Width Changes?” In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. 2018, pp. 488–497. URL: http://www.easychair.org/publications/paper/wPNs (cited on page 4).

[JS18c] Martin Jonáš and Jan Strejček. “On the complexity of the quantified bit-vector arithmetic with binary encoding.” In: *Information Processsing Letters* 135 (2018), pp. 57–61. DOI: 10.1016/j.ipl.2018.02.018 (cited on page 4).

[JS19] Martin Jonáš and Jan Strejček. “Q3B: An Efficient BDD-Based SMT Solver for Quantified Bit-Vectors.” In: *31st International Conference on Computer-Aided Verification – CAV 2019*. To appear. 2019 (cited on pages 4, 5, 100).

[Kin76] James C. King. “Symbolic Execution and Program Testing.” In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252 (cited on page 69).

[Knu09] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. 12th. Addison-Wesley Professional, 2009. ISBN: 0321580508, 9780321580504 (cited on page 44).

[KFB16] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. "Complexity of Fixed-Size Bit-Vector Logics." In: *Theory Comput. Syst.* 59.2 (2016), pp. 323–376. DOI: 10.1007/s00224-015-9653-1 (cited on pages 2, 29, 30, 35, 37, 104, 131).

[Koz06] Dexter Kozen. *Theory of Computation*. Texts in Computer Science. Springer, 2006. ISBN: 978-1-84628-297-3. DOI: 10.1007/1-84628-477-5 (cited on page 30).

[KLW15] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. "Under-approximating loops in C programs for fast counterexample detection." In: *Formal Methods in System Design* 47.1 (2015), pp. 75–92. DOI: 10.1007/s10703-015-0228-1 (cited on page 2).

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN: 978-3-540-74104-6 (cited on page 23).

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. "The UCLID Decision Procedure." In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. 2004, pp. 475–478. DOI: 10.1007/978-3-540-27813-9_40 (cited on page 1).

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665 (cited on page 69).

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on page 1).

[Loh12] Markus Lohrey. "Model-checking hierarchical structures." In: *J. Comput. Syst. Sci.* 78.2 (2012), pp. 461–490. DOI: 10.1016/j.jcss.2011.05.006 (cited on pages 33, 34).

[Lüc16] Martin Lück. "Complete Problems of Propositional Logic for the Exponential Hierarchy." In: *CoRR* abs/1602.03050 (2016). arXiv: 1602.03050. URL: http://arxiv.org/abs/1602.03050 (cited on pages 33, 34).

[MSS99] João P. Marques-Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability." In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433 (cited on page 39).

[Min01] Shin-ichi Minato. "Zero-suppressed BDDs and their applications." In: *STTT* 3.2 (2001), pp. 156–170. DOI: 10.1007/s100090100038 (cited on page 92).

[MB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers." In: *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*. 2007, pp. 183–198. DOI: 10.1007/978-3-540-73595-3_13 (cited on page 24).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on pages 1, 89, 106).

[MJ13] Leonardo Mendonça de Moura and Dejan Jovanovic. "A Model-Constructing Satisfiability Calculus." In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 2013, pp. 1–12. DOI: 10.1007/978-3-642-35873-9_1 (cited on page 104).

[Mrá+16] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiri Barnat. "SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration." In: *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. 2016, pp. 208–213. DOI: 10.1007/978-3-319-32582-8_14 (cited on page 2).

[Mrá+17] Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. "Optimizing and Caching SMT Queries in SymDIVINE - (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 2017, pp. 390–393. DOI: 10.1007/978-3-662-54580-5_29 (cited on page 5).

[Nav18] Peter Navrátil. "Adding Support for Bit-Vectors to BDD Libraries CUDD and Sylvan." Bachelor's thesis. Masaryk University, Faculty of Informatics, Brno, 2018. URL: https://is.muni.cz/th/lij5a/ (cited on page 89).

[NPB17] Aina Niemetz, Mathias Preiner, and Armin Biere. "Propagation based local search for bit-precise reasoning." In: *Formal Methods in System Design* 51.3 (2017), pp. 608–636. DOI: 10.1007/s10703-017-0295-6 (cited on pages 103, 104).

[Nie+18a] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. "Btor2 , BtorMC and Boolector 3.0." In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 2018, pp. 587–595. DOI: 10.1007/978-3-319-96145-3_32 (cited on pages 1, 114, 122).

[Nie+18b] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. “Solving Quantified Bit-Vectors Using Invertibility Conditions.” In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. 2018, pp. 236–255. DOI: 10.1007/978-3-319-96142-2_16 (cited on pages 2, 25, 27, 76, 86, 93, 103, 107, 111).

[OE11] Oswaldo Olivo and E. Allen Emerson. “A More Efficient BDD-Based QBF Solver.” In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*. 2011, pp. 675–690. DOI: 10.1007/978-3-642-23786-7_51 (cited on page 39).

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. “Automated Test Case Generation with SMT-Solving and Abstract Interpretation.” In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. 2011, pp. 298–312. DOI: 10.1007/978-3-642-20398-5_22 (cited on page 1).

[Pre17] Mathias Preiner. “Lambdas, Arrays and Quantifiers.” PhD thesis. Informatik, Johannes Kepler University Linz, 2017 (cited on page 41).

[PNB17] Mathias Preiner, Aina Niemetz, and Armin Biere. “Counterexample-Guided Model Synthesis.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. 2017, pp. 264–280. DOI: 10.1007/978-3-662-54577-5_15 (cited on pages 2, 25, 27, 93, 111).

[Rud93] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams.” In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*. 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029 (cited on page 44).

[Som15] Fabio Somenzi. “CUDD: CU Decision Diagram Package Release 3.0.0.” In: *University of Colorado at Boulder* (2015) (cited on pages 64, 89).

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. “From program verification to program synthesis.” In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, pp. 313–326. DOI: 10.1145/1706299.1706337 (cited on page 2).

[Udu+13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. “TRANSIT: specifying protocols with concolic snippets.” In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 287–296. DOI: 10.1145/2491956.2462174 (cited on page 25).

[WHM13] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. "Efficiently solving quantified bit-vector formulas." In: *Formal Methods in System Design* 42.1 (2013), pp. 3–23. DOI: 10.1007/s10703-012-0156-2 (cited on pages 2, 23, 24, 27, 29, 41, 93, 103, 111).

[ZWR16] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. "Deciding Bit-Vector Formulas with mcSAT." In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 249–266. DOI: 10.1007/978-3-319-40970-2_16 (cited on pages 1, 55, 104).

[ZWR17] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. "An Approximation Framework for Solvers and Decision Procedures." In: *J. Autom. Reasoning* 58.1 (2017), pp. 127–147. DOI: 10.1007/s10817-016-9393-1 (cited on pages 104, 113).

[Zel+18] Aleksandar Zeljic, Peter Backeman, Christoph M. Wintersteiger, and Philipp Rümmer. "Exploring Approximations for Floating-Point Arithmetic Using UppSAT." In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. 2018, pp. 246–262. DOI: 10.1007/978-3-319-94205-6_17 (cited on pages 104, 113).