Karlsruhe Institute of Technology

Solving difficult SMT instances using abstractions and incremental SMT solving

Bachelor's Thesis of

P. Samuel M. Teuber

at the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer: Prof. Dr. C. Sinz
Advisor: M.Sc. Marko Kleine Büning

15th May 2019 – 16th September 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.
Karlsruhe, Wednesday, 11th September 2019

(P. Samuel M. Teuber)

Acknowledgments

First of all, I would like to thank my advisors Marko Kleine Büning and Prof. Dr. Sinz for allowing me to pursue this exciting project - I very much enjoyed to work on this new approach for bitvector SMT solving. Furthermore, I'd like to thank Dominik Schreiber who contributed in sparking my interest in the field of logics and SAT solving.
On a more personal note I would like to thank Niklas Uhl who endured the task of reading my thesis draft and found several flaws in the text.

Abstract

Decision procedures for SMT problems based on the theory of bitvectors are a fundamental component in state-of-the-art software and hardware verifiers. In this work, we investigate whether abstractions can help in solving such problems. After a short survey of current abstraction techniques, we present a novel solving approach for the quantifier free bitvector theory (`QF_BV` in SMT-LIB) based on incremental SMT solving and abstraction refinement. We implement this approach in a prototype extending the SMT solver Boolector and evaluate its performance on the relevant benchmark subset of SMT COMP 2018. In comparison to Boolector, the new approach shows superior performance for unsatisfiable benchmark instances, while being inferior for satisfiable instances. Finally in future work, we propose various methods to further improve the performance, especially for satisfiable instances.

Zusammenfassung

Entscheidungsverfahren für SMT-Probleme in der Bitvektor-Theorie sind ein wesentlicher Bestandteil moderner Soft- und Hardware-Verifizierer. In dieser Arbeit untersuchen wir, ob Abstraktionsverfahren bei der Lösung von Bitvektor SMT-Problemen helfen können. Nach einem kurzen Überblick über aktuelle Abstraktionstechniken stellen wir einen neuartigen Lösungsansatz für die quantorenfreie Bitvektortheorie (`QF_BV` in SMT-LIB) vor, der auf inkrementellem SMT solving und Abstraction Refinement basiert. Wir implementieren diesen Ansatz in einem Prototyp zur Erweiterung des SMT-Solver Boolector und evaluieren seine Leistung anhand der relevanten Benchmark-Teilmenge der SMT COMP 2018. Im Vergleich zu Boolector zeigt der neue Ansatz eine bessere Leistung bei unerfüllbaren Benchmarkinstanzen, während er bei erfüllbaren Instanzen schlechter ist. Schließlich schlagen wir verschiedene Methoden vor, um die Leistung, insbesondere für erfüllbare Fälle, zukünftig zu verbessern.

Contents

1. Introduction

The satisfiability modulo theory (SMT) problem deals with deciding numerous fragments of the first-order logic constraint by some *Theory* T [3] and can be considered an extension of the satisfiability problem for propositional logic formulae [23]. A SMT-Theory T usually constraints the behaviour of certain uninterpreted functions or predicates of the first-order logic, but it may also syntactically constrain the language (e.g. by not allowing quantifiers). Today, solving SMT problems has become a discipline of its own with many solving techniques relying in one way or another on a SAT solver in the background. With SMT-LIB [2] a unified interface to codify SMT instances has been developed that is supported by most state-of-the-art SMT solvers.

Over the last decade a variety of approximation techniques have been introduced into the world of SMT solving and model checking. The objective of any such approximation techniques is to speed up the solving process thus reducing computational cost and avoiding exponential runtimes for common usecases. Generally speaking, the techniques can be categorized into over-approximation (or abstraction) techniques like lemmas on demand [10] on the one hand, and under-approximation techniques like the ones used in UCLID [11] or counterexample-guided abstraction refinement (CEGAR) [13] on the other hand. Usually, over-approximation techniques help in speeding up the solver's runtime for unsatisfiable SMT-instances, while under-approximation techniques help reducing the runtime of satisfiable instances [8].

Although abstraction techniques have been successfully used for the array theory [10] as well as the uninterpreted functions theory [27, 26], there is little work on using over-approximations for the quantifier free bitvector logic (`QF_BV` in SMT-LIB [2]) outside the approach taken with UCLID [11]. This thesis therefore takes up the topic of solving "difficult" bitvector SMT instances, like the ones provided in the *LLBMC family of benchmarks* [1], using abstractions.

Contributions Our contributions are twofold: First, we propose a theoretical framework allowing a simple proof of correctness for approximation techniques in SMT solving. Subsequently, we present an abstraction refinement technique for three functions of the `QF_BV` SMT-Theory (namely `bvmul`, `bvsdiv` and `bvsrem`), prove their correctness and evaluate their performance using the relevant benchmark subset of 2018's SMT competition [18] and a prototype implementation of the abstraction refinement approach based on Boolector [25]. We show that the abstraction approach at hand can solve 43 unsatisfiable instances more than Boolector which is a 30% improvement in comparison to the current number of unsatisfiable instances unsolved. Further, we find that the abstraction approach

currently performs worse than Boolector for satisfiable instances and propose various ideas for improving the runtime for this usecase in future work.

Outline After an introduction to the topic and a short survey of current abstraction and solving approaches in Chapters 2 and 3, we explain the pathway and intuition that led to the abstractions presented in Chapter 5. Finally, we present the implemented prototype in Chapter 6 and compare the prototype to Boolector's performance in Chapter 7.

2. Preliminaries

In this Chapter, we will introduce the foundations of Boolean algebra and SAT solving. Based on these techniques we will introduce SMT solving and present two predominant SMT solving approaches.

2.1. Basic boolean algebra and notation

The structure and notations of this Section are based on [23] and [7].

Definition 2.1.1 (Boolean variable, Atom, Literal)

- A *boolean variable* (represented by lowercase latin letters) is a variable which can be either **true** or **false** .
- An *atom* is a boolean variable.
- A *literal* l is a boolean variable x or its complement $\neg x$.

Definition 2.1.2 (Boolean formula)

- An atom, $\square$ (false) and $\blacksquare$ (true) are *boolean formulae.*
- If $\mathcal{F}$ is a *boolean formulae* so is $(\neg\mathcal{F})$.
- If $\mathcal{F}$ and $\mathcal{G}$ are *boolean formulae* so are $(\mathcal{F} \wedge \mathcal{G})$ and $(\mathcal{F} \vee \mathcal{G})$.

From hereon, we call the set of all Boolean formulae $\textsc{For}^0$.

We define $var(\mathcal{F})$ as the set of all variables appearing in $\mathcal{F}$. For the rest of this Chapter $\mathcal{F}$ and $\mathcal{G}$ will be assumed to be Boolean formulae.

Definition 2.1.3 (Interpretations and Models)
An *interpretation* I for $\mathcal{F}$ is a mapping $I\colon var(\mathcal{F}) \to \{0,1\}$. For some interpretation I the value of $I(\mathcal{F})$ is defined inductively through:

- $I(\square) = 0$ and $I(\blacksquare) = 1$.
- For $\neg\mathcal{G}$ the value is $(1 - I(\mathcal{G}))$.
- For $\mathcal{G}_1 \wedge \mathcal{G}_2$ the value is 1 iff $I(\mathcal{G}_1) = 1$ and $I(\mathcal{G}_2) = 1$, otherwise 0.
- For $\mathcal{G}_1 \vee \mathcal{G}_2$ the value is 1 iff $I(\mathcal{G}_i) = 1$ for either or both $i \in \{1,2\}$, otherwise 0.

Some interpretation I is a model for $\mathcal{F}$ iff $I(\mathcal{F}) = 1$ which we denote as $\mathcal{I} \vDash \mathcal{F}$.
A formula $\mathcal{F}$ is called satisfiable if there exists a model $\mathcal{I}$ for $\mathcal{F}$.

Logic formulae with connectors like $\iff$, $\implies$ etc. can similarly be defined as Boolean formulae however this is not strictly necessary as any boolean function can be represented by a formula only using the connectors described above. Parentheses may be omitted in which case $\neg$ takes precedence over $\wedge$ takes precedence over $\vee$.

Definition 2.1.4 (Conjunctions, Disjunctions and Clauses)
We call $\mathcal{F} \wedge \mathcal{G}$ a *conjunction* and $\mathcal{F} \vee \mathcal{G}$ a *disjunction*.
A *clause* is a disjunction of literals.
A formula is in *conjunctive normal form* (CNF) if the formula is a conjunction of clauses.
A formula is in *disjunctive normal form* (DNF) if the formula is a disjunction of conjunctions.

Example 2.1.5
$(a \vee \neg b \vee c) \wedge (d \vee \neg e) \wedge (g \vee b \vee c)$ is in CNF and
$(a \wedge \neg b \wedge c) \vee (d \wedge \neg e) \vee (g \wedge b \wedge c)$ is in DNF.

2.2. Satisfiability and SAT solving

The NP-complete problem of Satisfiability (SAT) concerns with whether a given Boolean formula $\mathcal{F}$ in CNF is satisfiable or not. A decision procedure for the SAT problem is called a SAT solver. For a proof on the NP-completeness of SAT the reader is referred to [16].

DPLL One of the first well-known algorithms for solving SAT problems is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [14] of which many variants are still used today. Generally speaking DPLL works by:

A) **Unit Propagation of single literal clauses**
Any literal which appears as the only literal in a clause may be assigned so that the clause evaluates to true

B) **Pure literal deletion**
Any literal which only appears as positive (or negative) literal may be fixed

C) **Case splitting**
If rules A) and B) can no longer be applied, some literal l is chosen and the problem is split up in the case where l is true and in the case where l is false.

CDCL Variants of the DPLL-algorithm have been extended by functionality to "learn" from contradictions encountered during search. This technique is called Conflict-Driven-Clause-Learning (CDCL) and works by adding a clause which makes sure to avoid the encountered contradiction during further search. For details on the methodology the reader is referred to [24].

Unsatisfiability Cores Another feature widely implemented in SAT solves is the ability to produce *Cores* for unsatisfiable instances. This allows the SAT solver to return the subset of formulae which was used to produce the contradiction making the instance at hand unsatisfiable. This feature is used in a wide range of applications including model checking, debugging specification and abstraction refinement [23].

Despite its theoretical hardness and the lack of known polynomial time algorithms for the SAT Problem, much progress has been made on practical *SAT solving* using variants of the DPLL-algorithm and CDCL. Among other applications those advances are particularly useful for the solving of so called SMT problems which are described in the next Section.

2.3. Satisfiability modulo theory

The previous Sections described various aspects of Boolean Algebra. For many applications, especially in software verification, however, a problem description in a more powerful language is much more desirable. For example, describing the manipulation of datastructures such as bitvectors and arrays is better comprehensible and a lot more concise than describing the manipulation of isolated bits in memory. This gives rise to the concept of *Satisfiability Modulo Theory* (SMT) which restricts first-order logic or higher-order logics to syntactic or semantic fragments offering a good trade-off between the language's expressiveness and the ability to automatically check an instance's satisfiability. Such fragments can then be decided by specialized decision-procedures exploiting properties of the specific sublanguage to enhance the solvers practical efficiency despite high worst-case computational complexity [3]. This Section will give an overview over SMT and then describe three specific theories in more detail.

2.3.1. The SMT language

In a first step, this Section introduces the many-sorted first-order logic as described in [3] with some inspiration from [7].

Definition 2.3.1 (Signature)
Given an infinite set $\mathbf{S}$ of *sort symbols* and an infinite set $\mathbf{X}$ of *variables* each assigned a sort $s \in \mathbf{S}$, a *signature* Σ consists of a tuple $\left(\Sigma^S, \Sigma^P, \Sigma^F, \int^P, \int^F\right)$ where:

- $\Sigma^S \subseteq \mathbf{S}$ is a set of sort symbols;
- Σ^P is a set of predicate symbols;
- Σ^F is a set of function symbols;
- $\int^P \colon \Sigma^P \to \left(\Sigma^S\right)^*$ is a mapping from predicate to predicate sort; and
- $\int^F \colon \Sigma^F \to \left(\Sigma^S\right)^+$ is a mapping from function to function sort.

Definition 2.3.2 (Arity and Rank)
For a function $g \in \Sigma^F$ with $\int^F(g) = \sigma_1 \cdots \sigma_n \sigma$ the *rank* of g is defined as $\sigma_1 \cdots \sigma_n \sigma$. We call n the *arity* of g (note that the arity may be 0).
For a predicate $p \in \Sigma^P$ with $\int^P(p) = \sigma_1 \cdots \sigma_n$ the *rank* of p is defined as $\sigma_1 \cdots \sigma_n$. We call n the *arity* of p (as before the arity may be 0 in which case $\int^P(p)$ is the empty word).

Definition 2.3.3 (Sorted Σ-terms)
A Σ-*term* of sort σ is either a variable $x \in \mathbf{X}$ of sort σ or an expression $g\,(t_1, \ldots, t_n)$ where $g \in \Sigma^F$ and $\int^F(f) = \sigma_1 \cdots \sigma_n \sigma$ with Σ-terms t_i of sort σ_i for every $i \in [\![1, n]\!]$.
For any $\sigma \in \Sigma^S$, we define $\textsc{Term}^{\Sigma}_{\sigma}$ as the set of all Σ-terms of sort σ.

Definition 2.3.4 (Atomic Σ-formulae, Σ-literals)
Atomic Σ*-formulae* are:

- The symbols $\square$ (false) and $\blacksquare$ (true);
- Expressions $t_1 \doteq t_2$ with t_1, t_2 Σ-terms of the same sort $\sigma \in \Sigma^S$; and
- Expressions $p\,(t_1, \ldots, t_n)$ with $p \in \Sigma^P$, $\int^P(p) = \sigma_1 \cdots \sigma_n$ and t_i of sort σ_i for every $i \in [\![1, n]\!]$.

In correspondence with the previous definitions of literals in Boolean Algebra a Σ-literal is an atomic Σ-formula ϕ or its complement $\neg\phi$.

Definition 2.3.5 (Σ-formulae)
Analogue to Boolean Algebra the Σ-*formulae* are defined inductively:

- Any Σ-literal is a Σ-formula.
- If ψ is a Σ-formula so are $(\neg\psi)$ and $\exists x\ \psi$ with $x \in \mathbf{X}$.
- If ψ and ϕ are Σ-formulae so are $(\psi \wedge \phi)$ and $(\psi \vee \phi)$.

Given this syntax the language's semantic can now be defined in a similar manner as for Boolean algebra using interpretations:

Definition 2.3.6 (Σ-interpretation)
For a signature Σ and a set $X \subseteq \mathbf{X}$ of variables with sorts in Σ^S a Σ-*interpretation* over X is a tuple $\mathcal{I} = \left(\mathcal{U}, \mathcal{I}^S, \mathcal{I}^X, \mathcal{I}^F, \mathcal{I}^P\right)$ where:

- $\mathcal{U} \neq \emptyset$ is the universe of all possible values;
- $\mathcal{I}^S \colon \Sigma^S \to 2^{\mathcal{U}}$[1] maps each sort σ_i to a domain $D_i \coloneqq \mathcal{I}^S\,(\sigma_i)$ of possible values for Σ-terms of this sort;
- $\mathcal{I}^X \colon X \to \mathcal{U}$ maps each variable $x \in X$ to a value $v \in \mathcal{U}$;
- $\mathcal{I}^F$ maps any function symbol $f \in \Sigma^F$ of rank $\int^F(f) = \sigma_1 \cdots \sigma_n \sigma_{n+1}$ to a function $f^{\mathcal{I}} \colon D_1 \times \cdots \times D_n \to D_{n+1}$; and

[1] $2^{\mathcal{U}}$ is the powerset of $\mathcal{U}$

- $\mathcal{I}^P$ maps any predicate $p \in \Sigma^P$ of rank $\int^P(p) = \sigma_1 \cdots \sigma_n$ to a truth function $p^{\mathcal{I}} \colon D_1 \times \cdots \times D_n \to \{0, 1\}$.

$\mathcal{I}^X$ must respect the sort σ_i of x (i.e., x of sort σ_i may only be mapped to values $v \in D_i$)

For terms t, variables v or predicates p, we denote their value according to a Σ-interpretation $\mathcal{I}$ as $t^{\mathcal{I}}$, $v^{\mathcal{I}}$ and $p^{\mathcal{I}}$.

Definition 2.3.7 (Substitution)
For some interpretation $\mathcal{I} = \left(\mathcal{U}, \mathcal{I}^S, \mathcal{I}^X, \mathcal{I}^F, \mathcal{I}^P\right)$ over variables $X \subseteq \mathbf{X}$ with $x \in X$ of type σ_i and $a \in D_i$, we define $\mathcal{I}\left[x \mapsto a\right]$ as the interpretation $\left(\mathcal{U}, \mathcal{I}^S, \overline{\mathcal{I}^X}, \mathcal{I}^F, \mathcal{I}^P\right)$ with:

$$\overline{\mathcal{I}^X}(v) = \begin{cases} a & v = x \\ \mathcal{I}^X(v) & \textit{otherwise} \end{cases}$$

Definition 2.3.8 (Σ-model)
A Σ-interpretation $\mathcal{I}$ is a Σ-*model* for some formula ϕ if $\mathcal{I}$ *satisfies* ϕ (i.e., $\mathcal{I} \vDash \phi$).
The *satisfies* relation $\vDash$ can be defined inductively:

- $\mathcal{I} \nvDash \square$ and $\mathcal{I} \vDash \blacksquare$.
- For Σ-terms t_1, t_2: $\mathcal{I} \vDash t_1 \doteq t_2$ iff $t_1^{\mathcal{I}} = t_2^{\mathcal{I}}$.
- For Σ-terms $t_1, \ldots, t_n$ and a suitable predicate $p \in \Sigma^P$:
  $\mathcal{I} \vDash p\left(t_1, \ldots, t_n\right)$ iff $p^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) = 1$.
- For some Σ-formula ψ: $\mathcal{I} \vDash \neg\psi$ iff $\mathcal{I} \nvDash \psi$.
- For Σ-formulae ϕ, ψ: $\mathcal{I} \vDash \phi \wedge \psi$ iff $\mathcal{I} \vDash \phi$ and $\mathcal{I} \vDash \psi$.
- For Σ-formulae ϕ, ψ: $\mathcal{I} \vDash \phi \vee \psi$ iff $\mathcal{I} \vDash \phi$ or $\mathcal{I} \vDash \psi$.
- For a Σ-formula ψ:
  $\mathcal{I} \vDash \exists x \colon \sigma_i \; \psi$ iff $\mathcal{I}\left[x \mapsto a\right] \vDash \psi$ for some $a \in D_i$.

Other logic connectors can be transformed into formulae in the form described above. In particular $\forall x \; \psi$ can be written as $\neg\exists x \; \neg\psi$. Furthermore $\exists x \; \psi$ can be written as $\exists x \colon \sigma \; \psi$ in order to state that x is of sort σ.

2.3.2. Theories

Given the generic many-sorted first order logic defined above, we can now define certain theories which constrain the interpretation of various predicates in Σ^P or functions in Σ^F to values in correspondence with the theory's desired behaviour. We describe 3 theories presented in [3] and explain how these theories can be intertwined yielding SMT-LIB theories [2].

Definition 2.3.9 (Σ-Theory)
A Σ-*Theory* is a tuple $T = (\Sigma, A)$ where Σ is a signature and A is a set-theoretical class of interpretations.

Definition 2.3.10 (T-interpretation, T-satisfiability, T-model, T-entails)

- Given a theory $T = (\Sigma, A)$ any Σ-interpretation $\mathcal{I}$ is a *T-interpretation* if $\mathcal{I} \in A$.
- A formula is *T-satisfiable* if it is satisfied by some T-interpretation.
- A T-interpretation which satisfies some formula ψ is a *T-model* for this formula (i.e., $\mathcal{I} \vDash_T \psi$).
- A set Φ of formulae *T-entails* a formula ψ (i.e., $\Phi \vDash_T \psi$) iff every T-interpretation that satisfies Φ also satisfies ψ.

While omitted in the definitions above for conciseness, it is worth to note that an interpretation $\mathcal{I}$ may be a T-model for some formula ψ even if $\mathcal{I}$ also defines values for functions, predicates or even sorts not in the signature of ψ. Among other benefits this can be useful when combining theories. From hereon, we denote the set of all Σ-formulae of some theory T as For_T^1.

Bitvectors: QF_BV The quantifier free (QF) theory of fixed size bit vectors (BV), abbreviated as QF_BV, concerns with the modeling of hardware and low-level software through bitvectors. For every $n \geq 1$ QF_BV contains a sort $\mathbf{BV}_n$ for bitvectors of length n. Any bit of such a vector may be either 0 or 1. In terms of function symbols the QF_BV theory contains extraction (typically written as $x[i]$ to extract a single bit and $x[i\colon j]$ to extract bits j to i with $i > j$) and concatenation (typically written as $x \circ y$) functions as well as a variety of well-known functions implemented in modern hardware and software (e.g. addition, multiplication, division, shifts, negation, and, or, xor etc.). In its general version QF_BV (encoded in binary) is NExpTime-complete [22].

Arrays: QF_ABV Adding arrays to the QF_BV theory results in a theory known as QF_ABV. The "stand-alone" extensional array theory contains as sorts $\mathbf{A}, \mathbf{I}$ and $\mathbf{E}$ (for **a**rray, **i**ndex and array **e**lements) however $\mathbf{I}$ and $\mathbf{E}$ are bitvectors in QF_ABV. In terms of function symbols a `read` as well as a `write` function are provided with $\int^F(\text{read}) = \mathbf{AIE}$ and $\int^F(\text{write}) = \mathbf{AIEA}$. The array theory is defined through 3 axioms which any signature must satisfy to support the (extensional) array theory:

$$\forall a\colon \mathbf{A}\ \forall i\colon \mathbf{I}\ \forall e\colon \mathbf{E}\ \text{read}\left(\text{write}\left(a, i, e\right), i\right) = e \tag{A1}$$

$$\forall a\colon \mathbf{A}\ \forall i\colon \mathbf{I}\ \forall e\colon \mathbf{E}\ i \neq j \implies \text{read}\left(\text{write}\left(a, i, e\right), j\right) = \text{read}\left(a, j\right) \tag{A2}$$

$$\forall a\colon \mathbf{A}\ \forall b\colon \mathbf{A}\ \left(\forall i\colon \mathbf{I}\ \text{read}\left(a, i\right) = \text{read}\left(b, i\right)\right) \implies a = b \tag{A3}$$

The non-extensional array theory can be obtained by dropping axiom (A3).

Uninterpreted Functions: QF_AUFBV The most simple theory imaginable is a theory containing nothing but arbitrary (uninterpreted) functions with equality. This theory can obviously be used to describe the application of functions on a wide range of objects. However, allowing arbitrary uninterpreted function symbols with bitvectors as parameters and outputs integrates the theory of uninterpreted functions (UF) into QF_ABV resulting in the QF_AUFBV theory. This theory can be axiomatized throught the *function congruence axiom* [27] where $\bar{x}$ and $\bar{y}$ are arbitrary bitvectors of bitwidth n:

$$\forall \bar{x}, \bar{y} \bigwedge_{i=1}^{n} x_i = y_i \implies f(\bar{x}) = f(\bar{y}) \tag{EUF}$$

2.4. SMT solving

Over time, a wide range of solvers has evolved making use of a variety of techniques to solve SMT formulae. While we refer the reader to [3] for a more complete overview of the various techniques used in SMT-solving the following Sections will give a brief overview of two predominant methodologies used in SMT-solving - namely Eager and Lazy SMT Solving

2.4.1. Eager SMT solving

Eager SMT solving is based on the idea of reducing a Σ-formula ϕ of some theory T into a Boolean algebra formula which can be solved by an off-the-shelf SAT solver.
This process can be described as a function tuple (e, t) where $e\colon \textsc{For}_T^1 \to \textsc{For}^0$ and t maps Boolean interpretations $\mathcal{I}^0$ to T-interpretations $\mathcal{I}_T^1$. For the eager SMT solving to work, functions (e, t) are needed such that $\mathcal{I}^0 \vDash e(\phi)$ iff $t(\mathcal{I}^0) \vDash \phi$.

A good example of the eager SMT solving approach is *bit blasting* which solves bitvector arithmetic problems from theories like QF_BV by assigning one Boolean variable for each bit appearing in an instance and then implements all functions used in the problem through the logic connectors available in Boolean algebra.

Example 2.4.1 (Addition)
Given two variables $x\colon \mathbf{BV}_2, y\colon \mathbf{BV}_2$ and the formula $11 \doteq add(x, y)$, the SMT problem can be transformed into Boolean algebra by writing out the addition:

$$\begin{aligned} 1 &\leftrightarrow x[0] \oplus y[0] && \textit{Calculating bit 0} \\ 1 &\leftrightarrow x[1] \oplus y[1] \oplus (x[0] \wedge y[0]) && \textit{Calculating bit 1} \end{aligned}$$

This problem can then be solved by an off-the-shelf SAT solver.

2.4.2. Lazy SMT Solving

It is easy to see that the eager approach described above can lead to very large Boolean formula sets. In particular complicated operations like multiplication or division instantly lead to a large number of formulae being added to the SAT solver instance. For this reason,

there exist a variety of approaches making use of SAT solvers to solve the overall structure (i.e., the abstraction explained below) of the problem while leaving the resolution of theory specific variable assignments to theory specific solvers.

Abstractions Given an infinite set Π of propositional variables, we define a mapping $(\cdot)^a$ which provides an injective mapping of atomic Σ-formulae to Π. For quantifier-free Σ-formulae, we can then define $(\neg\psi)^a = \neg(\psi)^a$, $(\psi \wedge \phi)^a = \psi^a \wedge \phi^a$ and $(\psi \vee \phi)^a = \psi^a \vee \phi^a$. Given some SMT formula Φ this allows us to give the overall structure (i.e., Φ^a) to an off-the-shelf (incremental) SAT solver which will return a model for Φ containing a set $S = \{\psi_1^a, \ldots, \psi_n^a\}$ of atomic Σ-formulae which need to be satisfied in this specific assignment.

Theory Solver A theory specific solver can then check whether the conjunction of S (specifically the conjunction of the original atomic Σ-formulae) has a satisfiable assignment or not. If it does, Φ is satisfiable, if not a clause

$$\bigvee_{\psi \in S} \neg\psi$$

is added to the SAT solver instance and another satisfying assignment of Φ^a can be searched to run the theory solver on. This can be repeated until the SAT solver either finds an assignment also satisfiable for the theory solver or finds the abstracted instance Φ^a and its added constraints to be unsatisfiable in which case Φ is unsatisfiable, too.

Algorithm To make the remarks made above clearer Algorithm 1 gives a glimpse on the inner workings of Lazy SMT solvers where *get_model* is a call to some (incremental) SAT solver and *check_sat*$_T$ is a call to the solver for theory T. Note that A^c returns the concrete atoms from their abstracted versions generated through ψ^a as previously explained.

Algorithm 1 A lazy SMT solving algorithm as presented in [3]

Require: ψ is a quantifier free Σ-formula of theory T
Ensure: output is `sat` if ψ is T-satisfiable, and `unsat` otherwise

```
F := ψ^a
loop
  A := get_model(F)
  if A = none then
    return unsat
  else
    μ := check_sat_T(A^c)
    if μ = sat then
      return sat
    else
      F := F ∧ ¬μ^a
    end if
  end if
end loop
```

3. Related Work

While bit blasting (as described in Chapter 2) is the predominant approach for solving the `QF_BV` theory and related theories, a multitude of techniques are being combined today in order to avoid bitblasting entire instances. In this Chapter, we give an overview of some of those techniques based on abstraction refinement.

3.1. Counterexample-guided abstraction refinement (CEGAR)

Counterexample-guided abstraction refinement (CEGAR) was first proposed in the field of software verification for the problem of verifying whether a given program P adhears to some specification ψ [13].
Given some program P containing variables $V = \{v_1, \ldots, v_n\}$ in domains $D_{v_1}, \ldots, D_{v_n}$ the set $D = D_{v_1} \times \cdots \times D_{v_n}$ is the set of states of P. Let p be some predicate, then $\text{Atoms}(p)$ is the set of atomic formulae in p and $\text{Atoms}(P)$ is the set of atomic formulae in P. If some program state $d \in D$ satisfies some predicate p, we write $d \vDash p$. A program defined this way can be directly transformed into a labeled Kripke-Structure defined as $\mathcal{M} = (S, I, R, L)$ with $S = D$, $I \subseteq D$, $R \subseteq S \times S$, $L \colon S \to 2^{\text{Atoms}(P)}$ where $L(d) = \{f \in \text{Atoms}(P) \mid d \vDash f\}$. The objective is then to compute whether $\mathcal{M} \vDash \psi$, that is, whether the P's Kripke-Structure $\mathcal{M}$ satisfies the specification ψ.

Abstractions An abstraction in the CEGAR sense is a surjection $h \colon D \to \hat{D}$. An abstraction h induces an equivalence relation on the set D of program states with $d \equiv_h e$ iff $h(d) = h(e)$. Note that this abstraction is an under-approximation of the problem as seen below. Given a Kripke-Structure $\mathcal{M}$ and an abstraction h the *abstracted Kripke-Structure* $\hat{\mathcal{M}} = \left(\hat{S}, \hat{I}, \hat{R}, \hat{L}\right)$ is defined through:

- $\hat{S} = \hat{D}$
- $\hat{d} \in \hat{I} \iff \exists d \in I \colon h(d) = \hat{d}$
- $\hat{R}\left(\hat{d}_1, \hat{d}_2\right) \iff \exists d_1, d_2 \in D \colon h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \wedge R(d_1, d_2)$
- $\hat{L}\left(\hat{d}\right) = \bigcup_{h(d)=\hat{d}} L(d)$

Theorem 3.1.1
Let h be an abstraction and ψ some specification. Given that for every atomic formula f in ψ and for all $d, d' \in D$ the property $(d \equiv_h d') \implies (d \vDash f \Leftrightarrow d' \vDash f)$ holds (we call

this "f respects h"), then:

(i) $\hat{L}\left(\hat{d}\right)$ is consistent for all abstract states $\hat{d}$ in $\hat{M}$

(ii) $\hat{\mathcal{M}} \vDash \psi \implies \mathcal{M} \vDash \psi$

Do note that while correctness (as defined by ψ) of the abstract model $\hat{\mathcal{M}}$ implies correctness of the original model $\mathcal{M}$ a model might still be correct if $\hat{\mathcal{M}} \nvDash \psi$.

Initial abstraction Upon initialization of the solving process an initial abstraction h is generated by grouping the variables $V = \{v_1, \ldots, v_n\}$ into disjoint variable clusters $V = VC_1 \dot{\cup} \cdots \dot{\cup} VC_n$. A variable cluster containing variable v_i contains any other variables v_j which appear in the same atomic formulae as v_i - this induces an equivalence relation relation on the variables. For each variable cluster $VC_i = \{v_{i_1}, \ldots, v_{i_k}\}$ an abstraction is defined through:

$$h_i(d_1, \ldots, d_k) = h_i(e_1, \ldots, e_k) \text{ iff for all atomic formulae } f$$
$$(d_1, \ldots, d_k) \vDash f \iff (e_1, \ldots, e_k) \vDash f$$

Handling counterexamples If the solver returns $\hat{\mathcal{M}} \vDash \psi$, Theorem 3.1.1 tells us that $\mathcal{M} \vDash \psi$. Otherwise the solver is assumend to return a counter example that can be checked on its correctness. More precisely it is necessary to check whether the counterexample is only possible in the abstracted structure $\hat{\mathcal{M}}$ or not. If the counterexample is caused by the abstraction and is therefore *spurious*, a refinement step is made detailing the previous abstractions and the solver will run once again on this less-abstracted version of the problem [13] until the counterexample is either correct or $\hat{\mathcal{M}} \vDash \psi$.

Since its introduction this approach of abstracting and refining has been used in wide range of applications including software verification [13], relational learning [12] and SAT based planning [15] as the core idea can be reused in most fields concerned with solving logic formulae.

3.2. Boolector and Lemmas on Demand

With Boolector [9] an approach to over-approximation called *Lemmas on Demand* was introduced to SMT solving. Boolector uses this extreme variant of lazy SMT solving for both solving array theory problems [10] and uninterpreted function theory problems [26]. This is particularly interesting as Boolector interleaves over- and under-approximation techniques as can be seen in Figure 3.1. Additionally Boolector makes heavy use of rewriting to solve easy bitvector theory instances - sometimes without using a SAT solver back-end at all.

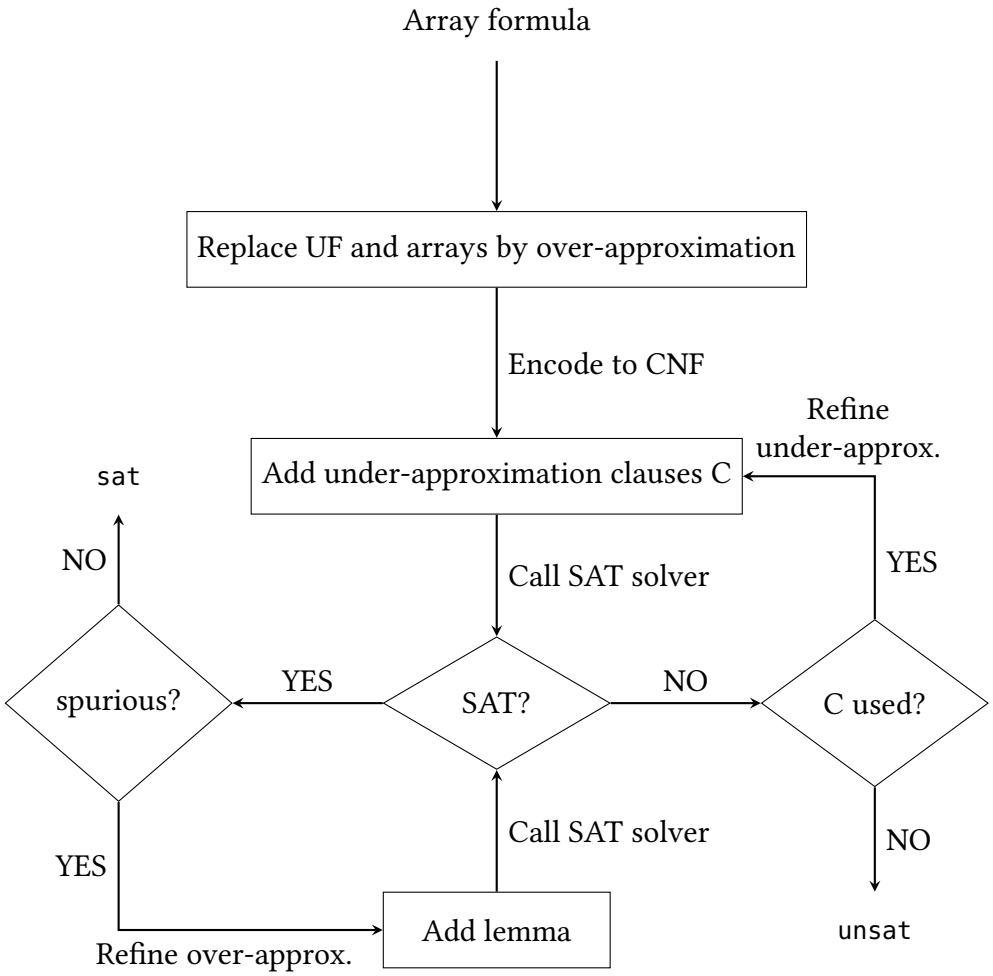


Figure 3.1.: Interleaving over- and under-approximation techniques in Boolector as presented in [8]

Rewriting SMT formulae passed to Boolector are rewritten in 3 levels [9]. In a first step very basic logic rules are applied during formula construction. In a second step global term substitution is performed on a topologically sorted DAG representation of the formula set. In a third and last step arithmetic normalization is performed.

Under-Approximation Boolector makes use of under-approximation on the CNF level by adding assumptions to the SAT solver instance [8]. Boolector restricts the *effective bitwidth* of a given bitvector to a smaller size which is then sign extended (or sometimes zero extended) to reach the original bitsize. Using a newly introduced assumption variable e this behaviour can be (de)activated as needed for each run through the SAT solvers assumption interface by adding/removing an activation clause e or $\neg e$. The additional constraints reduce the search-space size and thereby help to potentially speed up the solver's search. Furthermore, the additional constraints lead the solver towards smaller, usually better understandable models.

Lemmas on Demand Boolector uses over-approximation for solving the array [10] and uninterpreted function (UF) theories [27]. We will explain the idea behind this extreme variant of lazy SMT solving based on the uninterpreted function case.

For its initial abstraction every UF applications is replaced by a fresh bitvector variable. Afterwards the problem can be eagerly encoded as a SAT problem. If the SAT solver returns unsatisfiability the original problem is unsatisfiable, too. If on the other hand, the SAT solver returns a satisfying interpretation $\mathcal{I}$, we must now check whether the corresponding SMT interpretation $t(\mathcal{I})$ is consistent with the uninterpreted function's theory. More precisely, we must check whether for every function f its applications $f(\overline{x_1}), \ldots, f(\overline{x_{m_f}})$ are consistent with the (EUF) axiom.

If it is found that for two UF applications $t = f(a_1, \ldots, a_n)$ and $s = f(b_1, \ldots, b_n)$ the axiom is not respected (i.e., $a_i = b_i$ for all $i \in [\![1, n]\!]$ but $s \neq t$) an additional lemma encoding this constraint will be added. In the given counter-example, the choice of the uninterpreted function f on which the bitvectors are applied may depend on a certain number of `ITE` (if then else) conditions. To resolve this, the shortest paths p^s and p^t from the function application s (and t) to f are calculated and all `ite` conditions $c_0^s, \ldots, c_j^s$ ($c_0^t, \ldots, c_k^t$) on the paths evaluating to ■ under $t(\mathcal{I})$ as well as all `ite` conditions $d_0^s, \ldots, d_l^s$ ($d_0^t, \ldots, d_m^t$) on the paths evaluating to □ under $t(\mathcal{I})$ are collected. Remember that $t(\mathcal{I})$ is the SMT model corresponding to the Boolean Algebra model $\mathcal{I}$ returned by the underlying SAT solver. Using this information the following lemma is added to the SAT instance:

$$\left(\bigwedge_{i=0}^{j} c_i^s \wedge \bigwedge_{i=0}^{k} c_i^t \wedge \bigwedge_{i=0}^{l} \neg d_i^s \wedge \bigwedge_{i=0}^{m} \neg d_i^t \wedge \bigwedge_{i=0}^{n} a_i = b_i\right) \implies s = t.$$

This approach proved very effective in solving both the UF theory as well as the array theory. For its extension to arrays the reader is referred to [10]. While not relevant for this work [27] also explains how lemmas on demand can be used for lambda expressions. The technique can be further optimized by only refining those UF applications which are actually relevant for the current satisfying counterexample [26].

Overflow Detection Boolector implements efficiently encoded predicates for overflow detection in addition, substraction, multiplication and division [8]. Let $L(a)$ be the number of leading bits (zero or one) for some bitvector a. Given a signed multiplication instance $r = a * b$ where a and b are bitvectors of bitwidth n, an overflow occurs iff $L(a) + L(b) < n$ or $r[n] \oplus r[n-1]$ [20]. This result allows to check a signed multiplication for overflow issues by only calculating the first $n + 1$ multiplication bits in contrast to $2n$ bits for the naive encoding. Similar predicates are available for all basic arithmetic operations.

3.3. UCLID

UCLID [11] was one of the SMT solvers which introduced the interleaving of over- and under-Approximations using abstractions. Given an input formula ϕ the UCLID solver constructs an SMT under-approximation $\underline{\phi}$. The formula is usually generated by restricting variables to their sign-extended versions of a smaller bit size as previously seen in Boolector

(i.e., $v_n \cdots v_1$ becomes $v_m \cdots v_m v_{m-1} \cdots v_1$.) This formula $\underline{\phi}$ is then eagerly encoded and passed to a SAT solver. The SAT solver can produce one of two outcomes:

SAT In case the SAT solver finds a model such that $t(\mathcal{I}) \vDash \underline{\phi}$ the solver returns `sat` as $t(\mathcal{I}) \vDash \phi$. In this case UCLID can potentially speed up the SAT solver's runtime due to the reduced search space when solving the under-approximation $\underline{\phi}$

UNSAT In case the SAT solver returns an unsatisfiability result, UCLID uses the UNSAT-Core returned by the solver to extract the the formulae which produced the contradiction. Only based on these formulae (thereby leaving out all formulae of the instance that are not part of the contradiction) an over-approximation $\overline{\phi}$ is built. In contrast to $\underline{\phi}$ this over-approximation does not restrict the bitwidth of the input variables to some $m < n$, but allows the full bitwidth n. $\overline{\phi}$ is then passed to the SAT solver. If the SAT solver still returns unsatisfiability, UCLID returns `unsat`. In case the solver returns satisfiability, the under-approximation $\underline{\phi}$ is refined (usually increasing the bitwidth) and a second iteration begins.

For unsatisfiable instances UCLID takes advantage of cases where a small number of formulae can produce the contradiction and makes the SAT-solver only look at these formulae thereby potentially improving the solver's performance.

4. Solving “difficult” SMT instances

The “LLBMC Family of Benchmarks” is a benchmark family introduced in [1] containing SMT instances which are typically hard to solve for classical `QF_BV` solvers. The objective of this thesis is to investigate why certain instances of this benchmark family seem intrinsically hard to solve and whether abstractions coupled with incremental SMT solving could enable classic `QF_BV` solvers to decide some of the instances within reasonable time constraints. In this Chapter, we describe the pathway and intuitions which lead to the abstractions presented in Chapter 5, where the hasty reader may want to read on.

modmul A good example for the kind of benchmarks this family contains is the `modmul` benchmark which demands the solver to proof that for arbitrary bitvectors x, y and n:

$$x \doteq \texttt{mul}\,(y, n) \Rightarrow \texttt{srem}\,(x, n) \doteq 0$$

This is done by proving the unsatisfiability of the formula’s contradiction. While solvers like Boolector are able to solve the 8 bit case within a reasonable time span, the 32 bit case takes longer than 8 hours to solve insinuating an exponential runtime growth and making it a very hard problem to solve despite it’s briefness. This is particularly cumbersome as such basic results, which humans are usually able to categorize as correct within seconds, could be quite useful when deciding larger SMT problems in which this instance might happen to be embedded. We therefore started by analyzing this instance and searching for plausible abstractions using Boolector as basis for our experiments.

4.1. Naive decomposition

Probably the most naive approach to find an over-approximation of a given problem is to simply drop a certain number of clauses on the SAT-level before initiation of the solving process. This makes the solver ignore a certain number of clauses thereby potentially reducing the pathways a solver must take before finding a contradiction. This is of course provided we did not drop “too many” constraints and the instance is now satisfiable in which case we have to go back and add further constraints before another round of solving. Before diving any deeper into the problem, we therefore decomposed the And-Inverger-Graph [6] produced by Boolector for the `modmul` instance by splitting up the tree into it’s non-decomposable output nodes. We then investigated whether certain subsets of these nodes might already be enough to produce a contradiction thus proving unsatisfiability. This turned out not to be the case. On the contrary, all subsets evaluated were satisfiable and only when adding the very last output node the solver returned `unsat`. If at all the decomposition above only had a negative impact on the solver’s runtime.

4.2. Less naive decomposition

Given the rather useless results above, we could have already given up hope on decomposition being the key to success however a less naive way of decomposing the instance - namely adding the multiplication function bit by bit incrementally to the instance - had not yet been evaluated. Although this decomposition neither ended up improving the runtime performance of the solver the fact that it did not help is probably just as interesting for understanding the train of thought leading up to the successful abstraction.

The idea behind this second decomposition was to only add the least significant bit of the multiplication function on the instance's first run and later on incrementally add more and more bits of the multiplication logic. The intuition of this decomposition scheme was that this might lead to a contradiction earlier on as it's clear that the input formula is just as false for an 8 bit case as it is for a 32 bit case. This however did not work out the way it was intended as the upper bits were now free from any constraints and therefore could be set in any way necessary to find a counter-example. Only after the instance's last bit was added, the solver would return an unsatisfiability result and again with nothing but negative impact on the runtime needed.

4.3. The instance's core

At this point it became clear that using "just the formulae that are already there" could not significantly speedup the solver's runtime. This became even clearer looking at the UNSAT-Core produced by the underlying SAT solver: It turned out that some 90% of the instance's SAT clauses were in fact inside the UNSAT-Core and therefore (at least according to the SAT solver used) necessary to produce the desired contradiction. In particular this result suggested that the methodology used in UCLID [11] would not help in deciding this instance. Unfortunately - due to a lack of source, binaries and time - we were not able to test `modmul` on the UCLID solver. While it was not possible to show any kind of relation between the hardness of instances and their Core size in later (small scale) experiments it seemed like the only way to produce a contradiction faster would be through the addition of further (or other) information to the instance.

4.4. More information: values and intervals

The easiest way to add information is to predefine the result values of multiplication and/or remainder for certain well known values like $1, 0$ etc. Furthermore, for multiplications we can define intervals based on the factor's most significant bit and thereby define an interval for the multiplication's result. While those ideas will be described in more detail in the next Chapter it turned out that even those abstractions did not help with finding a solution for this particular instance. It's important to note these constraints were added in a first step before full multiplication was added afterwards if and only if the first step returned satisfiability.

4.5. More information: structure

In a last step, we then looked at giving the solver information on the relations between different functions. At this point it seemed clear that for any abstraction that would focus on the concrete values of the functions the solver would most likely have to try out a very large number of those values before producing the desired contradiction. The alternative was therefore to abstract away the actual values entirely and only look at what properties must be upheld between functions. Looking into the C++ standard [19], we find that the following property must be upheld for any modulo function application:

$$\texttt{mul}\,(n, \texttt{sdiv}\,(x, n)) + \texttt{srem}\,(x, n) = x$$

With minor modifications necessary for this abstraction to work for instances with varying bitwidths and overflows, this abstraction (detailed in the next Chapter) allowed solving the `modmul` instance for 32 bits in seconds instead of hours.

Later experiments showed that these value and interval based as well as the structure based abstractions not only helped in this case but also improved the solver's performance for a certain number of previously (within SMTCOMP's time constraints) unsolved benchmarks of SMTCOMP 2018 [18].

5. Refinement approach

In this Chapter, we present our abstraction procedure for the quantifier free bitvector theory (QF_BV). The approach substitutes applications of specific functions (here bvmul, bvsdiv and bvsrem) by *abstractions* defined on the QF_UFBV theory. During the solving process, an instance's abstractions are iteratively refined until the SAT solver either returns unsatisfiability or satisfiability with correct assignments. In Chapter 7 we will see that this approach outperforms Boolector's current results for unsatisfiable instances. After a theoretical definition of our abstraction methodology in Section 5.1 we present abstraction schemes for bvmul (Section 5.2), bvsdiv (Section 5.3) and bvsrem (Section 5.4).

5.1. Abstraction scheme

In the SMT-LIB standard [2] for QF_BV the functions examined in this work (i.e., bvmul, bvsdiv and bvsrem) support *overloading* in the sense that a single function symbol like bvmul supports multiple ranks. Multiplication for example is supported for any bitwidth. To simplify the explanations in the following Sections one can think of $\texttt{bvmul}^r$ as the bvmul operation of rank r thereby avoiding the issue of overloading.

Definition 5.1.1 (Approximation)
Given some theory $T = (\Sigma, A)$ and some function symbol $\texttt{op} \in \Sigma^F$ with $\int^F (\texttt{op}) = \sigma_1 \cdots \sigma_n \sigma$ and $n \geq 1$, a T-approximation for op consists of:

- a new uninterpreted function symbol $ap_{\texttt{op}}$ with $\int^F (\texttt{op}) = \int^F (ap_{\texttt{op}})$; and
- a mapping $\mathcal{A}_{\texttt{op}} \colon \textsc{Term}^{\Sigma}_{\sigma_1} \times \cdots \times \textsc{Term}^{\Sigma}_{\sigma_n} \to 2^{\textsc{For}^1_T}$.

A T-approximation can therefore be written as a tuple $(ap_{\texttt{op}}, \mathcal{A}_{\texttt{op}})$.

We can now define possible properties of a T-approximation. This will be useful to prove the correctness of our abstraction procedure later on

Definition 5.1.2 (Sound T-approximation)
Given some theory $T = (\Sigma, A)$ a T-approximation $(ap_{\texttt{op}}, \mathcal{A}_{\texttt{op}})$ is *sound* iff for all $\overline{x} \in Dom^1 (\mathcal{A}_{\texttt{op}})$ the following property holds:
For all T-interpretations $\mathcal{I}$ with $\mathcal{I} \vDash \mathcal{A}_{\texttt{op}} (\overline{x})$, it holds that $\mathcal{I} \vDash ap_{\texttt{op}} (\overline{x}) \doteq \texttt{op} (\overline{x})$.

Definition 5.1.3 (Complete T-approximation)
Given some theory $T = (\Sigma, A)$ a T-approximation $(ap_{\texttt{op}}, \mathcal{A}_{\texttt{op}})$ is *complete* iff for all $\overline{x} \in Dom (\mathcal{A}_{\texttt{op}})$ the following property holds:
For all T-interpretations $\mathcal{I}$ with $\mathcal{I} \vDash ap_{\texttt{op}} (\overline{x}) \doteq \texttt{op} (\overline{x})$, it holds that $\mathcal{I} \vDash \mathcal{A}_{\texttt{op}} (\overline{x})$.

[1] Dom is the domain of a given function. In this case $Dom (\mathcal{A}_{\texttt{op}}) = \textsc{Term}^{\Sigma}_{\sigma_1} \times \cdots \times \textsc{Term}^{\Sigma}_{\sigma_n}$

Example 5.1.4
For readability let $\mathtt{f} \coloneqq \mathtt{mul}^{\mathbf{BV}_2\mathbf{BV}_2\mathbf{BV}_2}$ be the 2-bit multiplication function as defined in QF_BV.

Example for a sound approximation:

$$\mathcal{A}_{\mathtt{f}}(x_1, x_2) \coloneqq \{ap_{\mathtt{f}}(x_1, x_2) \doteq 0, (x_1 \doteq 0 \vee x_2 \doteq 0)\}$$

Example for a complete approximation:

$$\mathcal{A}_{\mathtt{f}}(x_1, x_2) \coloneqq \{(x_1 \doteq 0 \vee x_2 \doteq 0) \implies ap_{\mathtt{f}}(x_1, x_2) \doteq 0\}$$

Example for a sound and complete approximation:

$$\mathcal{A}_{\mathtt{f}}(x_1, x_2) \coloneqq \{ap_{\mathtt{f}}(x_1, x_2) \doteq \mathtt{f}(x_1, x_2)\}$$

Essentially a sound T-approximation describes an under-approximation and a complete T-approximation describes an over-approximation of some function op. Using the notions defined above we can now define an abstraction scheme which iteratively refines an over-approximation until the abstraction converges into an exact description of the given function.

Definition 5.1.5 (Abstraction Scheme)
Given some theory $T = (\Sigma, A)$ and some function symbol $\mathtt{op} \in \Sigma^F$ of arity greater 0, a T-abstraction scheme is a finite totally ordered set of T-approximations

$$\mathcal{AS}_{\mathtt{op}} = \{\left(ab_{\mathtt{op}}, \mathcal{A}^1_{\mathtt{op}}\right), \ldots, \left(ab_{\mathtt{op}}, \mathcal{A}^k_{\mathtt{op}}\right)\}$$

where:

- For every $i \in [\![1, k]\!]$: $\mathcal{A}^i_{\mathtt{op}}$ is a complete T-approximation of op
- $\mathcal{C}_{\mathtt{op}}(\overline{x}) \coloneqq \left(\bigcup_{(\cdot, \mathcal{A}) \in \mathcal{AS}_{\mathtt{op}}} \mathcal{A}(\overline{x}) \right)$ is a sound T-approximation of op [2]

Lemma 5.1.6 (Completeness of Abstraction Schemes)
Given some T-abstraction scheme $\mathcal{AS}_{\mathtt{op}} = \{\left(ab_{\mathtt{op}}, \mathcal{A}^1_{\mathtt{op}}\right), \ldots, \left(ab_{\mathtt{op}}, \mathcal{A}^k_{\mathtt{op}}\right)\}$ with the properties defined above, $\mathcal{C}_{\mathtt{op}}$ is a complete T-approximation of op.

Proof. Let $\overline{x}$ be an arbitrary input vector for op.
For any T-interpretation $\mathcal{I}$ with $\mathcal{I} \vDash ap_{\mathtt{op}}(\overline{x}) \doteq \mathtt{op}(\overline{x})$ we know that by definition $\mathcal{I} \vDash \mathcal{A}^i_{\mathtt{op}}(\overline{x})$ for all $i \in [\![1, k]\!]$ as all approximations $\mathcal{A}^i_{\mathtt{op}}$ are complete.
Therefore,

$$\mathcal{I} \vDash \bigcup_{(\cdot, \mathcal{A}) \in \mathcal{AS}_{\mathtt{op}}} \mathcal{A}(\overline{x}) \text{ (i.e., } \mathcal{I} \vDash \mathcal{C}_{\mathtt{op}}(\overline{x})\text{)}$$

which implies that $\mathcal{C}_{\mathtt{op}}$ is a complete T-approximation, too.

[2] Just like all previous T-approximations $\mathcal{C}_{\mathtt{op}}$ is defined as $\mathcal{C}_{\mathtt{op}} \colon \textsc{Term}^{\Sigma}_{\sigma_1} \times \cdots \times \textsc{Term}^{\Sigma}_{\sigma_n} \to 2^{\textsc{For}^1_T}$ for a function symbol op of rank $\sigma_1 \cdots \sigma_n \sigma$

Lemma 5.1.7 (Soundness/Completeness through Implication)
Given some theory $T = (\Sigma, A)$ and a T-approximation $(ap_{\mathsf{op}}, \mathcal{A}_{\mathsf{op}})$:
If for all interpretations $\mathcal{I}$ and all $\overline{x} \in Dom\left(\mathcal{A}_{\mathsf{op}}\right)$

$$\mathcal{A}_{\mathsf{op}}\left(\overline{x}\right) \implies ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right)$$

then $(ap_{\mathsf{op}}, \mathcal{A}_{\mathsf{op}})$ is sound.
If for all interpretations $\mathcal{I}$ and all $\overline{x} \in Dom\left(\mathcal{A}_{\mathsf{op}}\right)$

$$ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right) \implies \mathcal{A}_{\mathsf{op}}\left(\overline{x}\right)$$

then $(ap_{\mathsf{op}}, \mathcal{A}_{\mathsf{op}})$ is complete.

Proof. The proof is based on Definitions 5.1.2 and 5.1.3.
Given for some formula the soundness (completeness) formula above holds for all $\mathcal{I}$ and $\overline{x}$:
For any interpretation $\mathcal{I}$ where $\mathcal{I} \nvDash \mathcal{A}_{\mathsf{op}}\left(\overline{x}\right)$ ($\mathcal{I} \nvDash ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right)$) the definition for soundness (completeness) is already fulfilled.
In case $\mathcal{I} \vDash \mathcal{A}_{\mathsf{op}}\left(\overline{x}\right)$ ($\mathcal{I} \vDash ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right)$) for some interpretation $\mathcal{I}$, then we know through the formula above that $\mathcal{I} \vDash ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right)$ ($\mathcal{I} \vDash \mathcal{A}_{\mathsf{op}}\left(\overline{x}\right)$) which implies that the approximation is, by definition, sound (complete).

Theorem 5.1.8 (Correctness of abstraction approach)
Let $T = (\Sigma, A)$ be some theory with $\mathsf{op} \in \Sigma^F$, $\int^F(\mathsf{op}) = \sigma_1 \cdots \sigma_n\sigma$ and $n \geq 1$.
Let further Φ be an arbitrary Σ-formula containing some function application $\mathsf{op}\left(\overline{x}\right)$.
Given some term $t \in \textsc{Term}_\sigma^\Sigma$ we define $\Phi\left[\mathsf{op}\left(\overline{x}\right) \mapsto t\right]$ as the formula where $\mathsf{op}\left(\overline{x}\right)$ is replaced by t in ϕ.
For any T-abstraction scheme $\mathcal{AS}_{\mathsf{op}}$ with function symbol ab_{op} the following property holds:
There exists a T-interpretation $\mathcal{I}_\Phi$ which is a T-model for Φ iff there exists a T-interpretation $\mathcal{I}_\mathcal{A}$ which is a T-model for

$$\Psi \coloneqq \Phi\left[\mathsf{op}\left(\overline{x}\right) \mapsto ab_{\mathsf{op}}\left(\overline{x}\right)\right] \wedge \bigwedge_{(\cdot,\mathcal{A}) \in \mathcal{AS}_{\mathsf{op}}} \mathcal{A}\left(\overline{x}\right)$$

Proof. The Theorem will be proved in 2 directions. For each direction we will construct a suitable interpretation given the premise interpretation.

$\Rightarrow$ Let $\mathcal{I}_\Phi$ be a T-model for Φ.
We build a model $\mathcal{I}_\mathcal{A}$ by extending $\mathcal{I}_\Phi$ so that $\mathcal{I}_\mathcal{A}\left(ab_{\mathsf{op}}\left(\overline{x}\right)\right)$ evaluates to $\mathcal{I}_\Phi\left(\mathsf{op}\left(\overline{x}\right)\right)$.
As $\mathcal{I}_\mathcal{A} \vDash ap_{\mathsf{op}}\left(\overline{x}\right) \doteq \mathsf{op}\left(\overline{x}\right)$, the completeness proof in Lemma 5.1.6 yields
$\mathcal{I}_\mathcal{A} \vDash \mathcal{A}_{\mathsf{op}}\left(\overline{x}\right)$.
Therefore

$$\mathcal{I}_\mathcal{A} \vDash \Phi\left[\mathsf{op}\left(\overline{x}\right) \mapsto ab_{\mathsf{op}}\left(\overline{x}\right)\right] \wedge \bigwedge_{(\cdot,\mathcal{A}) \in \mathcal{AS}_{\mathsf{op}}} \mathcal{A}\left(\overline{x}\right)$$

$\Leftarrow$ Let $\mathcal{I}_{\mathcal{A}}$ be a T-model for Ψ.
The abstraction scheme definition states that

$$\mathcal{I}_{\mathcal{A}} \vDash \bigwedge_{(\cdot, \mathcal{A}) \in \mathcal{AS}_{\mathsf{op}}} \mathcal{A}(\overline{x})$$

implies $\mathcal{I}_{\mathcal{A}} \vDash ap_{\mathsf{op}}(\overline{x}) \doteq \mathsf{op}(\overline{x})$ through the soundness property.
This implies that $\mathcal{I}_{\mathcal{A}} \vDash \Phi$.

Abstraction approach In the following Sections abstraction schemes for three function symbols of the `QF_BV` theory will be presented. In order to proof that the abstractions are actually valid we will:

(A) Proof the completeness of any approximation proposed

(B) Proof the soundness of each abstraction scheme

The abstractions can then be used to build a decision procedure like the one described in Algorithm 2. In a first step, the algorithm replaces all operations which should be refined by their abstracted uninterpreted functions. Afterwards the instance is reevaluated in a loop so long as the underlying SMT solver does not return `unsat` and the model returned is incorrect. In each round further constraints from the abstraction schemes are added to the instance. As shown in Theorem 5.1.8, (A) and (B) will be enough to prove that the abstraction schemes yield correct results when used in such a decision procedure.

Algorithm 2 Decision procedure for QF_BV abstractions. `ADD_CLAUSES` and `SAT` are calls to the underlying SMT solver.

Require: $\phi \in \text{FOR}^1_{QF\_BV}$
  abstractions$\leftarrow \langle\rangle$
  operations$\leftarrow \langle\rangle$
  for all op $\in \Sigma^F$ **do**
    if op needs refinement **then**
      for all op $(\overline{x})$ in ϕ **do**
        $\phi \leftarrow \phi\left[\text{op}\left(\overline{x}\right) \mapsto ab_{\text{op}}\left(\overline{x}\right)\right]$
        abstractions.push($\mathcal{AS}_{\text{op}}$)
        operations.push(op $(\overline{x})$)
      end for
    end if
  end for
  ADD_CLAUSES(ϕ)
  loop
    $r \leftarrow$SAT()
    if not r **then**
      print unsat
    else
      correct $\leftarrow$ **true**
      for all op $(\overline{x})$ in operations **do**
        if not op $(\overline{x})$ assignment is correct **then**
          correct $\leftarrow$ **false**
        end if
      end for
      if correct **then**
        print sat
      else
        for all $\mathcal{AS}$ in abstractions **do**
          ADD_CLAUSES($\mathcal{AS}$.pop()) {Add next approximation to for $\mathcal{AS}$ to instance}
        end for
      end if
    end if
  end loop

5.2. Abstracting `bvmul`

The abstraction scheme for `bvmul` is devided into four stages: The first stage describes the behaviour of `bvmul` for various common cases (like factors 0 and 1); the second stage defines intervals for the result value given the intervals of the multiplication factors; the third stage introduces relations between `bvmul` and other functions (specifically `bvsdiv` and `bvsrem`); the fourth stage finally adds full multiplication for certain intervals of the factors.

Overflow detection The `bvmul` function essentially behaves just like "regular" integer multiplication for any input value which does not produce an overflow. For many of the abstractions proposed in this Chapter it is therefore essential to detect overflows in order to filter out these special cases. As we have already seen in Section 3.2, Boolector supports a predicate to detect signed and unsigned multiplication overflows. The issue with this predicate is however that we have to calculate the first $w+1$ steps of multiplication given bitvectors of width w. This is rather impractical when we only even want to abstract the first w steps of multiplication. Therefore a predicate with no need for a multiplication unit is needed. Such a predicate without a multiplication unit, while complete, might not be sound - i.e., while every overflow might be detected, this predicate might detect more overflows than actually exist. The methodology used here is based on [21] where another approach for detecting overflows by counting the leading bits (ones or zeros) is proposed: The predicate ensures that there are at least $w+2$ leading bits for any multiplication of 2 negative numbers and at least $w+1$ leading bits for any other multiplication. While excluding a few cases with no overflows this predicate is never true for a pair of factors which results in an overflow. The corresponding predicate is defined in Definition 5.2.1 (for a multiplication instance `bvmul` (a, b) with a and b bitvectors of width w) and will be used on multiple occasions in the following Sections.

Definition 5.2.1 (*noov* predicate)
We define a Boolean function $noov : \{0,1\}^w \times \{0,1\}^w \to \{0,1\}$ as

$$
\begin{aligned}
noov\,(a,b) = &\bigvee_{n=0}^{w-1} \left(\bigwedge_{i=0}^{n} \neg a[w-i-1] \wedge \bigwedge_{i=n}^{w-1} \neg b[i] \right) \vee \\
&\bigvee_{n=0}^{w-1} \left(\bigwedge_{i=0}^{n} a[w-i-1] \wedge \bigwedge_{i=n}^{w-1} \neg b[i] \right) \vee \\
&\bigvee_{n=0}^{w-1} \left(\bigwedge_{i=0}^{n} \neg a[w-i-1] \wedge \bigwedge_{i=n}^{w-1} b[i] \right) \vee \\
&\bigvee_{n=0}^{w-2} \left(\bigwedge_{i=0}^{n} a[w-i-2] \wedge \bigwedge_{i=n}^{w-2} b[i] \wedge a[w-1] \wedge b[w-1] \right)
\end{aligned}
$$

5.2.1. Simple cases

For a multiplication instance $ab_{\texttt{bvmul}}(x, y)$ of factors x and y with bitwidth w we define the following constraints:

$$(x \doteq 0) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq 0) \tag{5.1}$$
$$(y \doteq 0) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq 0) \tag{5.2}$$
$$(x \doteq 1) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq y) \tag{5.3}$$
$$(y \doteq 1) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq x) \tag{5.4}$$
$$(x \doteq -1) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq -y) \tag{5.5}$$
$$(y \doteq -1) \Rightarrow \quad (ab_{\texttt{bvmul}}(x, y) \doteq -x) \tag{5.6}$$
$$noov(x, y) \Rightarrow \quad (\neg x[w-1] \wedge \neg y[w-1]) \Rightarrow (ab_{\texttt{bvmul}}(x, y) \geq 0) \tag{5.7}$$
$$\wedge\, (\neg x[w-1] \wedge y[w-1]) \Rightarrow (ab_{\texttt{bvmul}}(x, y) \leq 0) \tag{5.8}$$
$$\wedge\, (x[w-1] \wedge \neg y[w-1]) \Rightarrow (ab_{\texttt{bvmul}}(x, y) \leq 0) \tag{5.9}$$
$$\wedge\, (x[w-1] \wedge y[w-1]) \Rightarrow (ab_{\texttt{bvmul}}(x, y) > 0) \tag{5.10}$$

Equations (5.1) and (5.2) define the multiplication cases where one factor is zero and (5.3) as well as (5.4) define the cases where one factor is 1. Furthermore (5.5) and (5.6) define the negation cases.
Additionally we can make statements about the result's sign whenever we can be certain that no overflow is going to happen. For the cases where no overflow happens the sign behaviour of bitvector multiplication corresponds to the "common" sign behaviour and can therefore be split into 3 distinct cases:

- Both factors are non-negative producing a non-negative result (5.7)
- Both factors are negative producing a positive result (5.10)
- One of the factors is negative producing a non-positive result (5.8), (5.9)

Additionally all cases where one of the two factors is a power of 2 can be covered by constraints like (5.11) for all $i \in [\![1, w-1]\!]$ and for x and y symmetrically:

$$\bigwedge_{j \neq i} \neg x[j] \wedge x[i] \implies \left(umul\left(x_2^+, y_2^+\right) \doteq \texttt{shl}\left(y^+, i\right)\right) \tag{5.11}$$

where $umul$ is the unsigned multiplication function and x_2^+ as well as y_2^+ are positive, double bitwidth versions of x and y as detailed in the following Section. `shl` is the left shift function.

Completeness The completeness of this abstraction is a direct consequence of Lemma 5.1.7 as it can be checked that all formulae presented above (which specifically omitted any statements about difficult overflow cases) are implications of $ab_{\texttt{bvmul}}(x, y) \doteq \texttt{bvmul}(x, y)$.

5.2.2. Most significant digit based intervals

Using the factors' most significant digits, intervals of the factors can be defined which in turn can be used to assert intervals of the multiplication's result. In a first step the signed multiplication $r \coloneqq ab_{\texttt{bvmul}}(x, y)$ is transformed into its unsigned version with doubled bitwidth by using the absolute values:

$$
\begin{aligned}
x_2^+ \doteq \texttt{ITE(}&x[w-1],\\
&-\texttt{sext}(x, w)\\
&\texttt{sext}(x, w)\texttt{)}
\end{aligned}
\qquad
\begin{aligned}
y_2^+ \doteq \texttt{ITE(}&y[w-1],\\
&-\texttt{sext}(y, w),\\
&\texttt{sext}(y, w)\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
r_2' \doteq \texttt{ITE(}&x[w-1] \oplus y[w-1],\\
&-umul(x_2^+, y_2^+),\\
&umul(x_2^+, y_2^+)\texttt{)}
\end{aligned}
$$

ITE is the if-then-else function in SMT-LIB: If the first parameter is 1 the second parameter is returned, otherwise the third parameter is returned. sext is the sign extension function. By asserting equality of the multiplication result r and $r_2'[w-1:0]$ it is then possible to reason about the results of $r_2^+ \coloneqq umul(x_2^+, y_2^+)$ through bit shifting: If i is the most significant digit of x' then $2^i \leq x' < 2^{i+1}$. Therefore $2^i * y' \leq r_2^+ < 2^{i+1} * y'$. We now define a predicate $msd(x, i)$ which is true iff the most significant digit of x is i.

Definition 5.2.2 ($msd(x, i)$)
For some bitvector x of width w and an $i \in [\![0, w-1]\!]$ we define

$$msd(x, i) \coloneqq x[i] \wedge \bigwedge_{j=i+1}^{w-1} \neg x[j]$$

The previously presented results give rise to the following abstraction which must distinguish overflow from no-overflow cases. For this we will initally use a double bitwidth (i.e., $2 * w$ width) unsigned multiplication function as well as double bit width lower and upper bounds as defined below. We can then compare the necessary number of bits depending on the result of $noov$: If an overflow is possible we must compare the version with $2 * w$ bits, otherwise the w bit version can be used for comparison:

$$lower(a, b, n) \coloneqq \begin{cases} \texttt{ITE(}msd(a, 0), b, 0\texttt{)} & , n = 0 \\ \texttt{ITE(}msd(a, n), \texttt{shl}(b, n), lower(a, b, n-1)\texttt{)} & , else \end{cases}$$

$$upper(a, b, n) \coloneqq \begin{cases} \texttt{shl}(b, 1) & , n = 0 \\ \texttt{ITE(}msd(a, n), \texttt{shl}(b, n+1), upper(a, b, n-1)\texttt{)} & , else \end{cases}$$

$$
\begin{aligned}
&noov(x', y') \Rightarrow\\
&\quad lower(x_2^+, y_2^+, w)[w-1:0] \leq r_2^+[w-1:0] < upper(x_2^+, y_2^+, w)[w-1:0]\\
&noov(x', y') \Rightarrow\\
&\quad lower(y_2^+, x_2^+, w)[w-1:0] \leq r_2^+[w-1:0] < upper(y_2^+, x_2^+, w)[w-1:0]
\end{aligned}
$$

$$\neg noov(x', y') \Rightarrow lower(x_2^+, y_2^+, w) \leq r_2^+ < upper(x_2^+, y_2^+, w)$$
$$\neg noov(x', y') \Rightarrow lower(y_2^+, x_2^+, w) \leq r_2^+ < upper(y_2^+, x_2^+, w)$$

Note that while $lower$ and $upper$ seem to be recursive functions here they can be unrolled into consecutive `ITE` statements when adding the bounds to the instance to easen the solver's decision process.

Completeness Through the distinction between overflow and no-overflow cases the various equations can be regardes as "normal" multiplication disregarding overflows. Therefore, it can be checked that these constraints are direct implications of $ab_{\texttt{bvmul}}(x, y) \doteq \texttt{bvmul}(x, y)$. This approximation is complete according to Lemma 5.1.7.

5.2.3. Relations to other functions

Additionally to the previously described abstractions which all focused on relations between inputs and outputs of the specific function we can also look at relations between the given instruction invocation $ab_{\texttt{bvmul}}(\cdot, \cdot)$ and other invocations of the same or other instructions. This provides the solver with more high-level information and can therefore be useful in cases where relations between multiple instruction calls already lead to a contradiction without looking at the implementation details of the instructions.

For the multiplication instruction $ab_{\texttt{bvmul}}(x_2, y_2)$, with x_2 and y_2 the double bitwidth ($2 * w$) versions of x and y, we propose the following abstractions, which become particularly interesting when combined with similar abstractions for the `bvsrem` (Section 5.4) and `bvsdiv` (Section 5.3) function:

$$\begin{aligned} ab_{\texttt{bvmul}}(x_2, y_2) &\doteq ab_{\texttt{bvmul}}(y_2, x_2) \\ x_2 \doteq 0 \vee y_2 &\doteq ab_{\texttt{bvsdiv}}(ab_{\texttt{bvmul}}(x_2, y_2), x_2) \\ y_2 \doteq 0 \vee x_2 &\doteq ab_{\texttt{bvsdiv}}(ab_{\texttt{bvmul}}(x_2, y_2), y_2) \end{aligned}$$

For every bit width $w' < 2 * w$ which appears in a given problem instance and its abstractions we can further assert that

$$ab_{\texttt{bvmul}}(x_2, y_2)[w' - 1 : 0] \doteq ab_{\texttt{bvmul}}(x_2[w' - 1 : 0], y_2[w' - 1 : 0])$$

and

$$ab_{\texttt{bvmul}}(x_2, y_2)[w' - 1 : 0] \doteq ab_{\texttt{bvmul}}(y_2[w' - 1 : 0], x_2[w' - 1 : 0])$$

and for

$$x' \coloneqq \texttt{sext}\left(x_2\left[\left\lfloor\frac{w'}{2}\right\rfloor : 0\right], w' - \left\lfloor\frac{w'}{2}\right\rfloor\right)$$
$$y' \coloneqq \texttt{sext}\left(y_2\left[\left\lfloor\frac{w'}{2}\right\rfloor : 0\right], w' - \left\lfloor\frac{w'}{2}\right\rfloor\right)$$

we assert that

$$ab_{\texttt{bvmul}}(x', y') \doteq ab_{\texttt{bvmul}}(y', x')$$

as well as

$$y' \doteq 0 \vee x' \doteq ab_{\texttt{bvsdiv}}\,(ab_{\texttt{bvmul}}(x', y'), y')$$

and

$$x' \doteq 0 \vee y' \doteq ab_{\texttt{bvsdiv}}\,(ab_{\texttt{bvmul}}(x', y'), x')$$

Essentially all these relations between various multiplication and division applications are all based on properties defined in the C standard [19] for multiplication and division.
The only challenge of this abstraction is to formulate the constraints so that they are even complete for overflow cases.
A naive approach would simply encode the constraints in single bitwidth w. This however, turns out to be problematic as the properties listed above do not hold for overflow cases that (according to the C standard) result in undefined behaviour and most certainly not in above constraints being respected. Therefore, we use an approach with doubled bitwidth $2 * w$ while encoding the constraints.

Completeness The completeness of this abstraction is a direct consequence of all assertions being well-known properties for machine multiplication and division also defined in the C standard. As we avoid all overflow cases through the use of doubled bitwidth, the properties hold for any input combination.

5.2.4. Full multiplication

In a last step full multiplication on a per-interval basis is added as constraint. We assume a SMT instance containing some multiplication `bvmul` (x, y). If the instance is still satisfiable after having been passed through refinement steps 5.2.1 to 5.2.3, the solver returns a counterexample with assignments for x and y. We then look up the most significant bit i of x's assignment and assert that:

$$msd\,(x, i) \implies ab_{\texttt{bvmul}}\,(x, y) \doteq \texttt{bvmul}\,(x, y)$$

Completeness and Soundness For a multiplication of bitwidth w the approximation is complete and it even becomes sound once this assertion has been made for all $i \in [\![0, w-1]\!]$. It is also to see that the maximum number of necessary refinement steps is bounded by w.

5.3. Abstracting `bvsdiv`

Just like multiplication, `bvsdiv` is a rather costly function in terms of formulae needed for its representation. Furthermore, concepts very similar to those used for abstracting `bvmul` can be reused as an abstraction approach for `bvsdiv`. We therefore present a very similar four step abstraction approach for `bvsdiv` in this Section.
Note that `bvsdiv` can only overflow in the case of dividing the minimum integer (10^{w-1} in binary) by -1. In SMT-LIB this overflow returns the minimum integer.

5.3.1. Simple cases

For some division $\texttt{bvsdiv}\,(x, y)$ of bitwidth w a certain number of simple cases can be encoded as a first abstraction step:

$$(y \doteq 0) \implies ab_{\texttt{bvsdiv}}\,(x, y) \doteq \texttt{ITE}\,(a < 0, 1, x) \tag{5.12}$$
$$(y \doteq 1) \implies ab_{\texttt{bvsdiv}}\,(x, y) \doteq x \tag{5.13}$$
$$(y \doteq x) \implies ab_{\texttt{bvsdiv}}\,(x, y) \doteq 1 \tag{5.14}$$
$$(y \doteq -1) \implies ab_{\texttt{bvsdiv}}\,(x, y) \doteq -x \tag{5.15}$$
$$-y < x < y \implies ab_{\texttt{bvsdiv}}\,(x, y) \doteq 0 \tag{5.16}$$

The first assertion (5.12) implements the SMT-LIB standard for divisions by zero. The other assertions cover various simple cases of division with (5.15) also covering the overflow case mentioned above (this is because $-(1 \circ 0^{w-1}) = 1 \circ 0^{w-1}$). Furthermore we can again make use of cases where y is a power of two by asserting that for all $i \in [\![1, w-1]\!]$:

$$\bigwedge_{j \neq i} \neg x\,[j] \wedge x\,[i] \implies \left(udiv\left(x^+, y^+\right) \doteq \texttt{ashr}\left(x^+, i\right)\right) \tag{5.17}$$

With $udiv$ being the unsigned division function and x^+ as well as y^+ being the positive versions of x and y as detailed in the next Section. `ashr` is the arithmetic right shift function.

Completeness The approximation is complete by Lemma 5.1.7 as the formulae above are implications of $ab_{\texttt{bvsdiv}}\,(x, y) \doteq \texttt{bvsdiv}\,(x, y)$.

5.3.2. Most significant digit based intervals

In correspondence to the abstraction approach for `bvmul`, we can make use of the most significant bit of the divisor y for some division $\texttt{bvsdiv}\,(x, y)$. The division problem will first be transformed into its unsigned version:

$$x^+ \doteq \texttt{ITE(}x[w-1],\ -x,\ x\texttt{)} \qquad y^+ \doteq \texttt{ITE(}y[w-1],\ -y,\ y\texttt{)}$$
$$r' \doteq \texttt{ITE(}x[w-1] \oplus y[w-1],\ -udiv(x^+, y^+),\ udiv(x^+, y^+)\texttt{)}$$

Just like for `bvmul` we then assert that

$$ab_{\texttt{bvsdiv}}\,(x, y) \doteq r'$$

so we can then assert constraints for $r^+ \doteq udiv(x^+, y^+)$:

$$lower(a,b,n) \coloneqq \begin{cases} 0 & ,n=0 \\ \texttt{ITE(}msd(b,n), \texttt{ashr}\,(a,n+1)\,, lower(a,b,n-1)\texttt{)} & ,else \end{cases}$$

$$upper(a,b,n) \coloneqq \begin{cases} a & ,n=0 \\ \texttt{ITE(}msd(b,n), \texttt{ashr}\,(a,n)\,, upper(a,b,n-1)\texttt{)} & ,else \end{cases}$$

$$lower(x^+, y^+, w) < r^+ \leq upper(x^+, y^+, w)$$

This assertion defines an interval for the value of r^+ (and through r^+ for the value of $ab_{\texttt{bvsdiv}}\,(x,y)$) depending on the most significant digit of y.

Completeness The completeness of this abstraction is a direct result of the formulae above being an implication of $ab_{\texttt{bvsdiv}}\,(x,y) \doteq \texttt{bvsdiv}\,(x,y)$.

5.3.3. Relations to other functions

Due to the difficulties of overflows and relation assertions explained in Subsection 5.2.3, double bitwidth variables x_2 and y_2 will be used in the following assertions[3]. Note however that as explained earlier the division itself only overflows in a single case already covered in Subsection 5.3.1.
For divison the following two assertions are being added (again in accordance with the C standard):

$$\begin{aligned} ab_{\texttt{bvsdiv}}\,(x,y) \doteq & ab_{\texttt{bvsdiv}}\,(x_2,y_2)\,[w-1:0] \\ x_2 \doteq & ab_{\texttt{bvmul}}\,(ab_{\texttt{bvsdiv}}\,(x_2,y_2)\,,y_2) + ab_{\texttt{bvsrem}}\,(x_2,y_2) \end{aligned}$$

Completeness The proof is analog to the proof in Subsection 5.2.3.

5.3.4. Full division

In a last step (also parallel to `bvmul`) division is added one interval at a time as explained in Subsection 5.2.4 for the multiplication case. In this case the intervals are based on the value of the dividend x. As for `bvmul`, the soundness of this abstraction scheme is a direct consequence of the assertions made in this step.

5.4. Abstracting `bvsrem`

Due to its rareness in benchmarks[4] only a single abstraction layer has been added for this function. Once again with double bitwidth as explained in Section 5.2.3, we assert the

[3] These variables are sign extended

[4] On the one hand its rareness makes it harder to evaluate the performance of abstractions on the other hand abstractions are less likely to have a big impact on the overall performance of the solver.

relations between bvsrem and other functions:

$$\begin{aligned} ab_{\texttt{bvsrem}}\,(x,y) \doteq & ab_{\texttt{bvsrem}}\,(x_2,y_2)\,[w-1:0] \\ x_2 \doteq & ab_{\texttt{bvmul}}\,(ab_{\texttt{bvsdiv}}\,(x_2,y_2)\,,y_2) + ab_{\texttt{bvsrem}}\,(x_2,y_2) \end{aligned}$$

In the following refinement step the full remainder constraint

$$ab_{\texttt{bvsrem}}\,(x,y) \doteq \texttt{bvsrem}\,(x,y)$$

is added.
The proof of completeness for the first abstraction is analog to the proof in Subsection 5.2.3. The proof of soundness of the abstraction scheme for bvsrem is trivial after the second assertion as $ab_{\texttt{bvsrem}}\,(x,y) \doteq \texttt{bvsrem}\,(x,y) \implies ab_{\texttt{bvsrem}}\,(x,y) \doteq \texttt{bvsrem}\,(x,y)$.

6. Implementation

As the focus of this work lay on researching which abstractions are effective in reducing the solvers runtime and not so much on building an improved solver, the abstraction refinement procedure is built as a layer on top of Boolector. In order enable a fast evaluation of the abstraction approaches, the prototype is implemented in Python. To increase readibility, we will refer to the prototype as "Ablector" which stands for *Ab*stracted Boo*lector*

Generally speaking there were 2 pathways we could have taken for the experiments: Either implement the abstraction refinement procedures directly into an existing SMT solver or build a layer on top of some SMT solver which then implements the abstraction refinement procedure. The big advantage of an implementation directly within an SMT solver would have been the performance enhancements through optimizations and integrations only possible from within (e.g. working directly inside the over- and under-approximation loop of Boolector described in Section 3.2). The disadvantage however would have been a much slower evaluation process due to the enhanced complexity when implementing new features in a project similarly scaled to Boolector.

Architecture While Boolector offers a Python Interface (pyboolector) on its own, this Python interface is difficult to extend when it comes to parsing SMT-LIB files as pyboolector moves directly into C-level parsing when evaluating SMT-LIB files and only returns to the Python level when the solver's result can be returned. Therefore another SMT-LIB parser was necessary to allow for the abstraction refinement to be implemented in Python. This resulted in the architecture presented in Figure 6.1 which will be explained below.

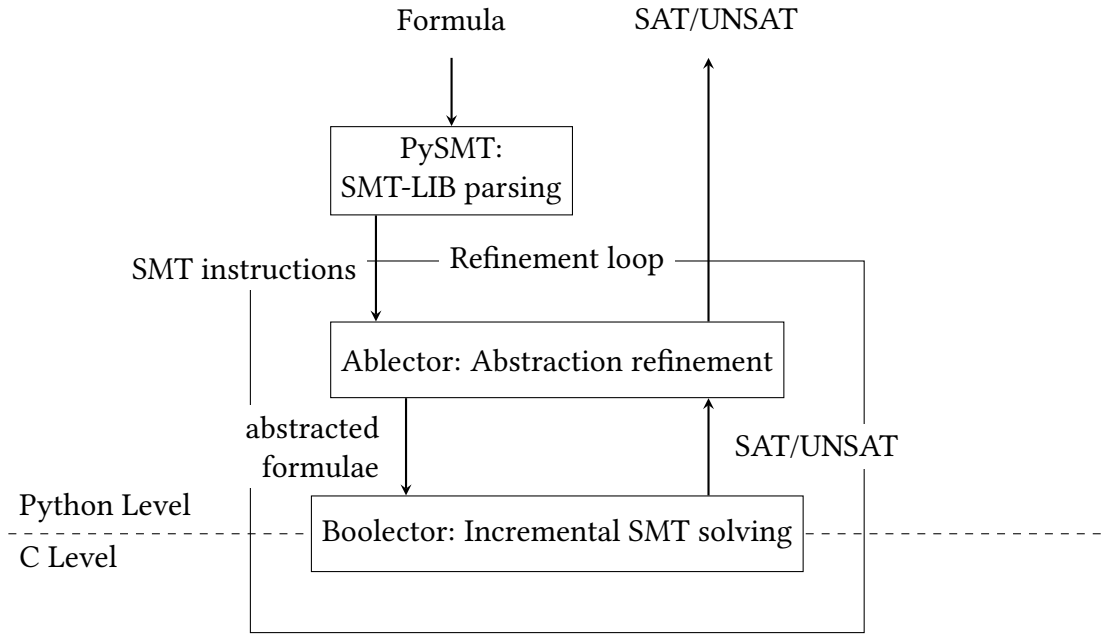


Figure 6.1.: The overall architecture of the prototype implemented

6.1. PySMT

The PySMT Python library [17] is used as a parser. The library allows SMT-LIB parsing and integrates interfaces for various SMT-solvers including Boolector. The library made it possible to enhance the Boolector interface in a way that allowed the implementation of the abstraction refinement procedure.
The downside of this library is its failing to parse a few benchmarks containing certain bitshift operations and the relatively slow parsing time in comparison with Boolector's parser[1].

6.2. Boolector

As underlying SMT solver Boolector [9] is used as it won both the `QF_ABV` and `QF_BV` main track at SMT-COMP 2018 [18]. The solver integrates with PySMT through its Python interface pyboolector and supports uninterpreted functions. For the implementation of the refinement loop the solver's python interface is extended through a subclass rewriting certain methods of pyboolector (like the method for multiplication and the method for signed division and remainder) in order to implement the abstraction procedures. As PySMT integrates the invocation of pyboolector's methods into its own parsing procedure this methodology resulted in a setup allowing to easily test out abstractions.

[1]Please note that this result was found using some very basic tracing techniques and therefore might not give a full picture of PySMT's parsing performance. In any case the parsing times of Boolector and PySMT seemed to differ sufficiently for this to potentially have an impact on the performance comparisons later on.

As already mentioned earlier Boolector makes heavy use of various optimization techniques before bit blasting. Oftentimes Boolector can actually solve SMT instances entirely without bit blasting. As these optimization techniques are rather quick in runtime, we actually run them on every instance before the abstraction refinement procedure to check if these procedures already find the instance to be unsatifiable. Only if the instance is not yet found to be unsatisfiable by these procedures, we continue and run our abstraction procedure on the original (that is, not the optimized) version of the SMT instance.

6.3. Abstraction node managment

Upon invocation of a method like `Mul`$(\cdot, \cdot)$ the pyboolector interface returns an object representing the result of the multiplication function which can again be passed into other pyboolector methods. The extending subclass keeps a list of so called *abstraction nodes*. Upon invocation of a rewritten method like `Mul`$(\cdot, \cdot)$ a new *abstraction node* is added to this list. The *abstraction node* of some function (e.g. `Mul`$(\cdot, \cdot)$) contains the entire logic necessary in the abstraction refinement steps. Among other methods any *abstraction node* must therefore contain the following methods:

- `isExact`()
  Returns whether the given node is already sound.
- `isCorrect`()
  Returns whether the current assignment of the given node is correct (i.e., a correct result of the function given the current input assignments).
- `refine`()
  Prepares the assertions necessary for the next refinement step.
- `doAssert`()
  Adds the assertions prepared in `refine`() to the solver instance.

This allows an easy abstraction of any function and makes the abstraction managment of the various functions independent from each other. The refinement loop can then call these methods on each *abstraction node* as needed.
Once an abstraction node returns that it is exact (i.e., both complete and sound), it is removed from the list of *abstraction nodes* and no longer visited in the refinement steps.

7. Evaluation

In the following Sections, the solving time with (*Ablector*) and without (*Boolector*) abstraction is compared. For details on the experimental setup as well as measures taken to secure the reproducibility of the results see Appendix A.

7.1. Time measurements

Like we already mentioned in the Section 6.1, the parsing engine of PySMT is notably slower than the parsing engine of Boolector. Comparing the real time (i.e., clock time) of the two competitors would therefore result in a very biased performance comparison as we are trying to benchmark the abstractions and not the parsing.
We therefore decided to only compare the CPU clock time of certain operations. Specifically, we compare the CPU clock time[1] of Boolector's SMT-LIB `check-sat` call against the summed up CPU clock time of all invocations to rewritten procedures in Ablector including its own `check-sat` call. This will produce a more realistic comparison of the abstraction's performance. Note that we are over-approximating the time Ablector takes in this comparison as we are adding the processing time for abstracted functions (e.g. $\texttt{Mul}(\cdot,\cdot)$) which are not considered for Boolector. However, those time measurements are in most cases negligible in comparison to the time for the `check-sat` call.
This time measurement will usually be referred to as `satpart` as it is essentially measuring the time the solver takes to produce an (un)sat result without parsing.

7.2. Benchmarks set

Initially, the benchmark set described in Section 4 was evaluated. In order to avoid *overfitting* the abstractions for this benchmarks, we later on evaluated the abstraction approach using a subset of the benchmarks used at SMT-COMP 2018 [18]. Specifically, we extract all benchmarks containing `bvmul`, `bvsdiv` and `bvsrem` function applications with the most appearances in the benchmark set being `bvmul` instructions. The benchmark set contains 15340 unsatisfiable instances and 4605 satisfiable instances. The focus of this work lies on the unsatisfiable instances. 76 unsatisfiable and 79 satisfiable benchmarks had to be omitted due to problems with the PySMT parser explained in Section 6.1.

[1] We chose the CPU clock time as it can be measured through the same unified interface in both Python and C with comparable results. As the tasks in this program Section are heavily CPU-bound with very little IO operations taking place, this measurement can still be considered realistic.

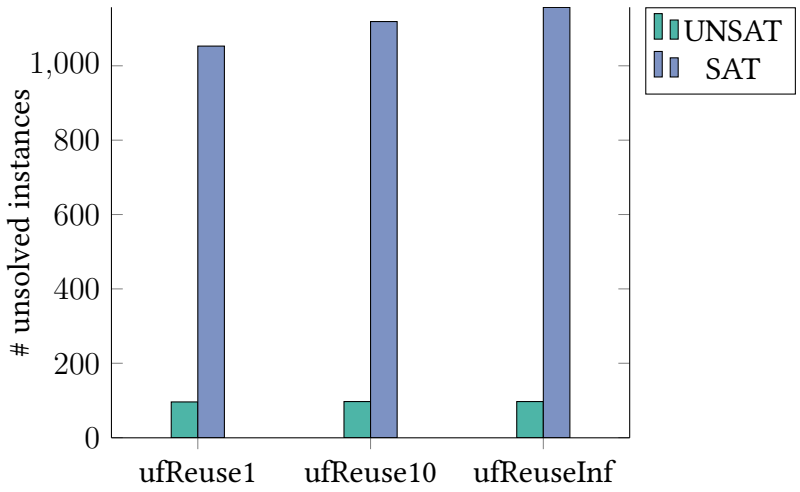


Figure 7.1.: Number of unsolved instances (both SAT and UNSAT) for fresh UF on every appearance (ufReuse1), fresh UF on every tenth appearance (ufReuse10) and the same UF for all appearances (ufReuseInf)

7.3. Reuse of uninterpreted functions

In our abstraction scheme any abstracted function is replaced by an application of some uninterpreted function (UF). For instances containing multiple invocations of the same abstracted function we can choose whether to reuse the same uninterpreted function for every appearance, or whether to use a "fresh" uninterpreted function for each appearance. This decision potentially has a big impact on the overall performance of the solver: On the one hand using a fresh function for each appearance reduces the number of Lemmas necessary for Boolector to bring the function results in a consistent state. On the other hand using the same function every time puts in place another - potentially useful - constraint on the function's results (i.e., it makes sure that the results are at least consistent for the same input even though they might still be wrong).

Figure 7.1 gives an overview of the number of unsolved instances with a fresh UF for each appearance (ufReuse1), the same UF for all appearances (ufReuseInf) and a fresh UF for every tenth appearance (ufReuse10). We can see that the ufReuse1 variant performs best and solves the largest number of instances. We therefore chose to procede with ufReuse1 for the further analysis.

7.4. Unsatisfiable Instances

The benchmark set considered contained a total of 15264 unsatisfiable instances. It is worth noting that 9404 of those instances are solved by Boolector without the use of a SAT solver at all leaving 5860 instances to be solved by the SAT solver. A first comparison of Boolector and Ablector is given in Figure 7.2 and 7.3. While the plots show that for certain instances Ablector seems to be slower than Boolector, we can also see a number of instances for which Ablector finds a solution while Boolector times out. Figure 7.3

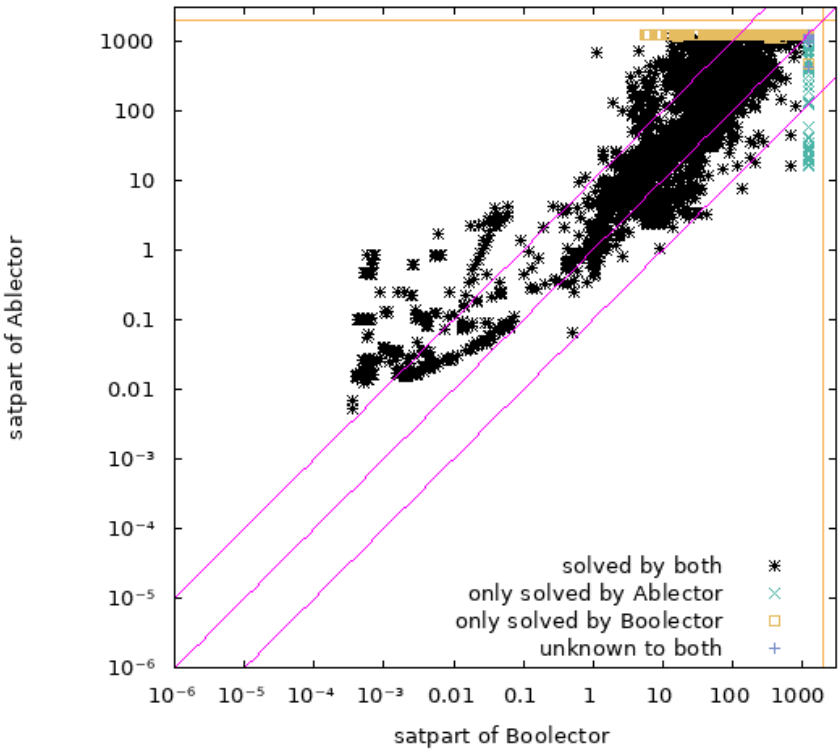


Figure 7.2.: `satpart` of Boolector vs `satpart` of Ablector in seconds for unsatisfiable instances

only shows the instances which had a runtime of more than 1s for either solver, thereby removing all time differences irrelevant to us.
Although we can very well evaluate the differences in running time in the two Figures, the plots lack essential information on the number of benchmarks solved by either solver. A better analysis of the number of instances (un)solved by Boolector and Ablector can be found in Table 7.1: Here we see that Ablector, using the abstractions presented above, is able to solve 43 instances more than Boolector. Note that in numbers (not looking at overlap etc.) this is about 30% of the instances for which Boolector times out.

		Boolector		
		unsolved	solved	
Ablector	unsolved	80	16	96
	solved	59	15109	15168
		139	15125	15264

Table 7.1.: Number of unsatisfiable instances solved by Boolector and Ablector

For any appearance of the `bvmul` and `bvsdiv` function within an instance we furthermore tracked the final abstraction level. Abstraction level 0 corresponds to the simple cases, level 1 corresponds to the msd based intervals, level 2 corresponds to the relations between

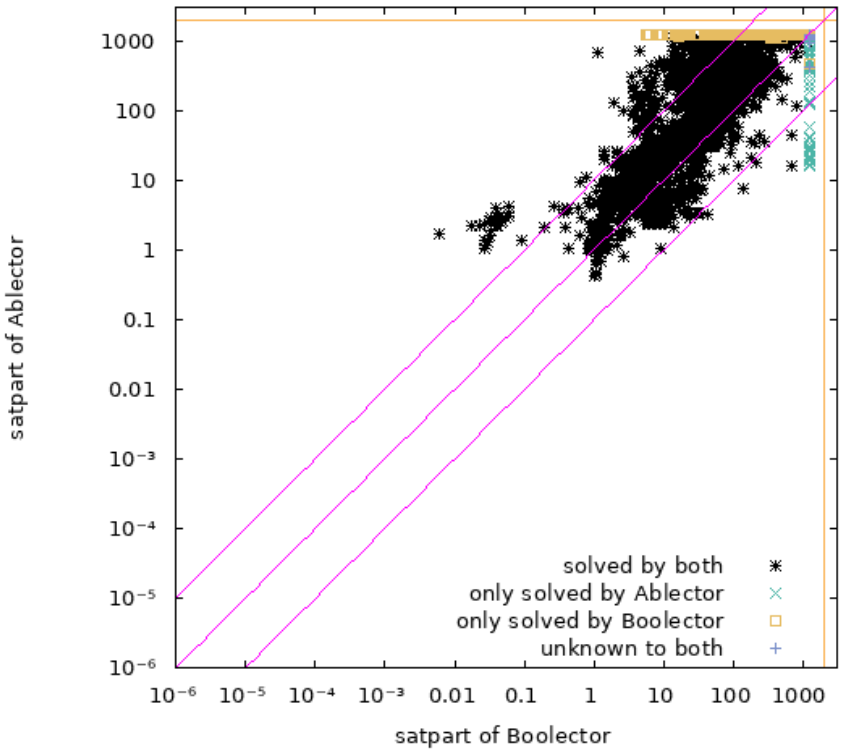
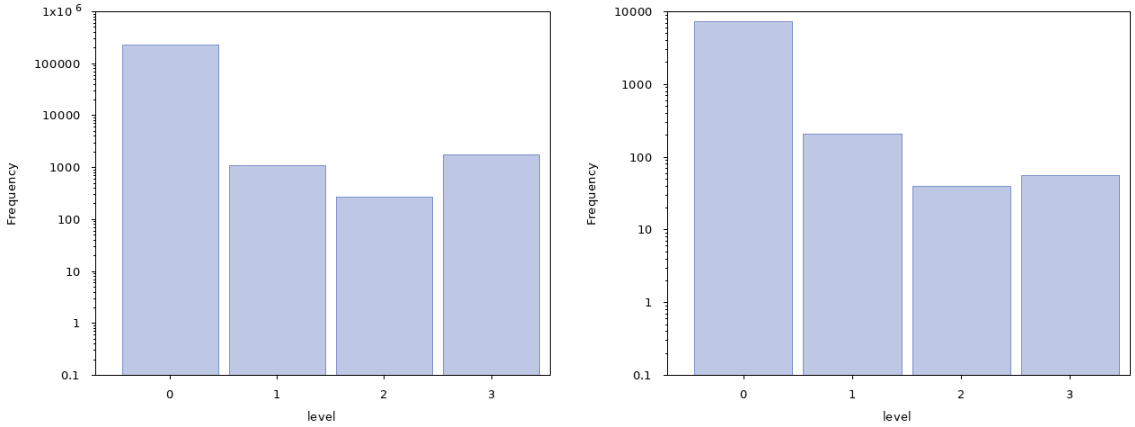


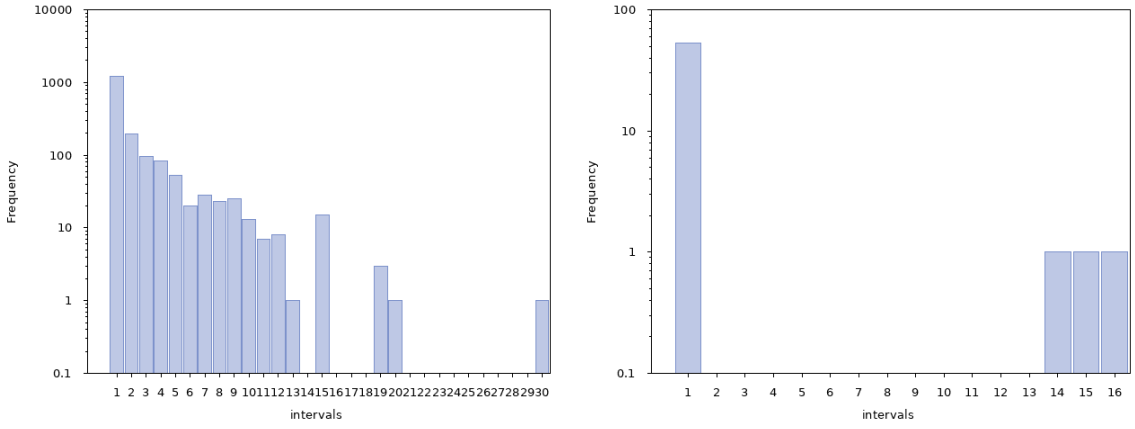
Figure 7.3.: `satpart` of Boolector vs `satpart` of Ablector in seconds for unsatisfiable instances with satpart larger $1s$ in either dimension

functions and level 3 corresponds to the interval-wise full multiplication/division. If abstraction level 3 was reached for some instance, we further track the number of intervals added. This is of interest as it gives an estimate on the necessity of each refinement step: For example, if a refinement level would never appear as the final refinement step for an unsatisfiable instance, it is very unlikely that this abstraction is of much use. However looking at Figure 7.4, we see that the final abstraction levels are somewhat distributed across the various abstraction steps and that all abstractions therefore help in solving certain instances. Further, Figure 7.5 shows that for many instances full multiplication for a single interval (or very few intervals) are enough to produce the unsat result. This implies that the incremental, last refinement level also helps in solving certain instances.



(a) `bvmul` (b) `bvsdiv`

Figure 7.4.: Final abstraction level of function applications: 0 are simple cases, 1 are bit shifts, 2 are UF relations and 3 is the interval-wise full multiplication/division step.



(a) `bvmul` (b) `bvsdiv`

Figure 7.5.: Number of intervals added for multiplication/division in the final abstraction step.

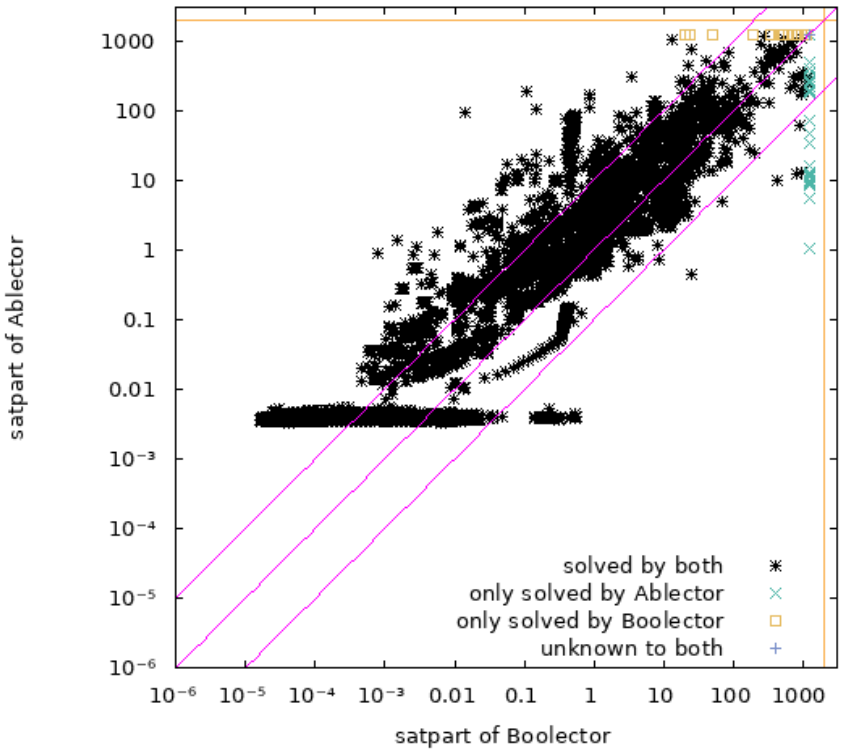


Figure 7.6.: `satpart` of Boolector vs `satpart` of Ablector in seconds for satisfiable instances

7.5. Satisfiable Instances

For satisfiable instances on the other hand, Abletor's performance is worse than Boolector's: As we can see in Figure 7.6 and Table 7.2, Boolector is able to solve a lot of instances Ablector cannot currently solve and the running time Ablector takes for the solved instances cannot make up for this flaw. It is also worth noting that, while about 500 timed out instances got stuck in the first refinement round, the rest of the timed out instances are evenly distributed across all refinement rounds.

While bounding the running time of each refinement round by an upper limit could avoid the problem of instances getting stuck in a certain step, we expect that the abstraction scheme's performance could further be improved in future work by integrating the abstractions directly into a solver like Boolector instead of building them as a layer on top. This would allow to make better use of the under-approximation techniques that are completely ignored for most abstraction steps in the current abstraction scheme.

		Boolector		
		unsolved	solved	
Ablector	unsolved	579	474	1053
	solved	73	3400	3473
		652	3874	4526

Table 7.2.: Number of satisfiable instances solved by Boolector and Ablector

8. Conclusion

In this work, we introduced a novel approach to solving quantifier free bitvector problems in SMT-LIB's `QF_BV` theory. The approach is based on abstraction methodologies previously used for various other problems in logic and specifically in SMT. On the one hand, we presented numerous abstractions for 3 comparatively costly funcitons of the bitvector theory, on the other hand, we proposed a simple theoretical framework allowing a proof of correctness for the presented abstractions. While we saw that the presented approach performs better than Boolector in deciding unsatisfiable bitvector problems, solving 43 instances more, the implemented prototype is not yet competitive for satisfiable instances. This is of course in some way a natural result, as over-approximations usually improve the solver runtime on unsatisfiable (and not on satisfiable) instances.

Future Work With the abstraction's correctness proved and the abstraction's performance evaluated through the current prototype, one could now implement the abstraction scheme directly into a SMT solver like Boolector making use of interleaved under- and over-approximations. We expect that this might enhance the solver's performance sufficiently to be competitive for both satisfiable and unsatisfiable instances. Additionally, a time limit for each refinement round could be introduced in order to avoid cases where the solver gets stuck in some specific refinement step. For this, a detailed parameter analysis will be necessary to find a time limit that keeps the abstraction steps effective while avoiding dead ends. Finally, it is left to investigate whether a *don't care reasoning* strategy similar to the one used for Lemmas on Demand in Boolector [26] could improve the abstraction refinement procedure's performance.

Bibliography

[1] Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo, eds. *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. Publication series B, Report B-2017-1. University of Helsinki, Department of Computer Science, 2017.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[3] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11.

[4] Armin Biere. "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017". In: *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Marijn Heule, and Matti Järvisalo. Vol. B-2017-1. Department of Computer Science Series of Publications B. University of Helsinki, 2017, pp. 14–15.

[5] Armin Biere. *runlim*. Website. URL: http://fmv.jku.at/runlim.

[6] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[7] Thierry Boy de la Tour and Mnacho Echenim. *Éléments de Logique pour le cours de 2 ème année Ensimag: Fondements de Logique pour l'Informatique*. 2015.

[8] Robert Brummayer. "Efficient SMT solving for bit vectors and the extensional theory of arrays". PhD thesis. Johannes Kepler University of Linz, 2010. ISBN: 978-3-85499-707-8.

[9] Robert Brummayer and Armin Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 174–177. DOI: 10.1007/978-3-642-00768-2_16.

[10] Robert Brummayer and Armin Biere. "Lemmas on Demand for the Extensional Theory of Arrays". In: *JSAT* 6.1-3 (2009), pp. 165–201. URL: https://satassociation.org/jsat/index.php/jsat/article/view/74.

[11] Randal E. Bryant et al. "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.

[12] Arun Chaganty et al. "Combining Relational Learning with SMT Solvers Using CEGAR". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 447–462. ISBN: 978-3-642-39799-8.

[13] Edmund M. Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[14] 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.

[15] Nils Froleyks, Tomas Balyo, and Dominik Schreiber. "PASAR—Planning as Satisfiability with Abstraction Refinement". In: *Twelfth Annual Symposium on Combinatorial Search*. 2019.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[17] Marco Gario and Andrea Micheli. "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms". In: *SMT Workshop 2015*. 2015.

[18] Matthias Heizmann et al. *SMT COMP 2018*. Website. URL: http://smtcomp.sourceforge.net/2018/.

[19] *ISO14882:2011(E) C++*. Standard. Geneva, CH: International Organization for Standardization, Sept. 2011.

[20] Michael J. Schulte et al. "Combined Unsigned and Two's Complement Saturating Multipliers". In: *Proceedings of SPIE - The International Society for Optical Engineering* 4116 (Sept. 2000). DOI: 10.1117/12.406496.

[21] Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013. ISBN: 0-321-84268-5. URL: http://www.hackersdelight.org/.

[22] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. "On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width". In: *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*. 2012, pp. 44–56. URL: http://www.easychair.org/publications/paper/145348.

[23] João P. Marques-Silva and Sharad Malik. "Propositional SAT Solving". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 247–275. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_9.

[24] 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. URL: https://doi.org/10.1109/12.769433.

[25] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector 2.0”. In: *JSAT* 9 (2014), pp. 53–58. URL: https://satassociation.org/jsat/index.php/jsat/article/view/120.

[26] Aina Niemetz, Mathias Preiner, and Armin Biere. “Turbo-charging Lemmas on demand with don’t care reasoning”. In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, pp. 179–186. DOI: 10.1109/FMCAD.2014.6987611.

[27] Mathias Preiner, Aina Niemetz, and Armin Biere. “Lemmas on Demand for Lambdas”. In: *Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013*. Ed. by Malay K. Ganai and Alper Sen. Vol. 1130. CEUR Workshop Proceedings. CEUR-WS.org, 2013. URL: http://ceur-ws.org/Vol-1130/paper_7.pdf.

A. Reproducibility

Software For alle experiments a modified version of `Boolector 3.0.1-pre` is used. More specifically we modified commit `f689fbbfe820392d35e26be368f9d87d2dbdb037` so that we could measure the time of the `check-sat` instruction. This can be found in branch `sat-time-measure` of `https://github.com/samysweb/boolector`. As underlying SAT-solver Lingeling [4] with version `bcj 78ebb8672540bde0a335aea946bbf32515157d5a` is used. All software packages were compiled using the provided cmake scripts which have the highest optimization levels enabled using gcc in version `(Ubuntu 5.4.0-6ubuntu1 16.04.10) 5.4.0 20160609`.
For the final experiments presented in Chapter 7 Ablector is used in the version available in commit 79584caeb4b7ea27ac3e80153b167c36d434232e at `https://github.com/samysweb/ablector`.

Machine All experiments were executed on a cluster of 20 identical compute nodes each housing 2 Intel Xeon E5430 @ 2.66GHz CPUs and a total of 32GB of RAM. The SMT benchmark files were stored on a RAID system connected to the cluster.

Benchmark execution 2 jobs were run in parallel on each compute node with the timeout set to 1200 seconds this posed no caching issues as they were run on seperate CPU sockets [1]. For time surveillance and measurements we used the runlim utility [5]. All benchmarking scripts and the log results can be obtained at `https://github.com/samysweb/BA-experiments`.

[1] Early on we ran up to 8 experiments on a single node to make use of the available cores however this seemed to produce caching issues slowing down the experiment times

B. List of Figures