Approximations and Abstractions for Reasoning about Machine Arithmetic

ALEKSANDAR ZELJIĆ
Approximations and Abstractions for Reasoning about Machine Arithmetic

Aleksandar Zeljić  
aleksandar.zeljic@it.uu.se  

October 2016

Division of Computer Systems  
Department of Information Technology  
Uppsala University  
Box 337  
SE-751 05 Uppsala  
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Aleksandar Zeljić 2016  
ISSN 1404-5117  
Printed by the Department of Information Technology, Uppsala University, Sweden
Safety-critical systems rely on various forms of machine arithmetic to perform their tasks: integer arithmetic, fixed-point arithmetic or floating-point arithmetic. Machine arithmetic can exhibit subtle differences in behavior compared to the ideal mathematical arithmetic, due to fixed-size of representation in memory. Failure of safety-critical systems is unacceptable, because it can cost lives or huge amounts of money, time and effort. To prevent such incidents, we want to formally prove that systems satisfy certain safety properties, or otherwise discover cases when the properties are violated. However, for this we need to be able to formally reason about machine arithmetic. The main problem with existing approaches is their inability to scale well with the increasing complexity of systems and their properties. In this thesis, we explore two alternatives to bit-blasting, the core procedure lying behind many common approaches to reasoning about machine arithmetic.

In the first approach, we present a general approximation framework which we apply to solve constraints over floating-point arithmetic. It is built on top of an existing decision procedure, e.g., bit-blasting. Rather than solving the original formula, we solve a sequence of approximations of the formula. Initially very crude, these approximations are frequently solved very quickly. We use results from these approximations to either obtain a solution, obtain a proof of unsatisfiability or generate a new approximation to solve. Eventually, we will either have found a solution or a proof that solution does not exist. The approximation framework improves the solving time and can solve a number of formulas that the bit-blasting cannot.

In the second approach, we present a novel method to reason about the theory of fixed-width bit-vectors. This new decision procedure is called mcBV and it is based on the model constructing satisfiability calculus (mcSAT). The procedure uses a lazy representation of bit-vectors and attempts to avoid bit-blasting altogether. It is able to reason about bit-vectors on both bit- and word-level, leveraging both Boolean constraint propagation and native arithmetic reasoning. It also features a greedy explanation generalization mechanism and is capable of more general learning compared to existing approaches. mcBV is able to reason about bit-vectors with sizes that significantly exceed the usual 32, 64 and 128 bits. Evaluation of mcBV shows an improvement in performance (compared to bit-blasting) on several classes of problems.
Acknowledgments

I want to thank my advisers Philipp Rümmer and Christoph M. Wintersteiger for their guidance, endless patience and support.

I also want to thank my colleagues who have made working at the IT department interesting and fun. Special thanks go to all the members of the Real-Time Systems Group, for all the coffee breaks, interesting discussions, inspiration and encouragement.

To my friends, old and new, far and near, thank you for putting up with my disappearances and radio silence when a deadline comes due. Your company, advice and support have been invaluable to me.

Finally, I want to thank my family for their unfaltering love and support.
List of Papers

This thesis is based on the following papers

I An Approximation Framework For Decision Procedures —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer. 
Accepted for publication in Journal of Automated Reasoning.

Based on:
Approximations For Model Construction —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer. 
Automated Reasoning — 7th International Joint Conference, 

II Deciding Bit-Vector Formulas Using mcSAT —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer. 
Theory and Applications of Satisfiability Testing - 19th International 

Comments on my participation:
I am the main author of all the papers included in this thesis.
# Contents

1 Introduction ......................................................... 3  
   1.1 Research questions ........................................... 4

2 An example .......................................................... 7  
   2.1 Analyzing the execution ....................................... 8
   2.2 Choosing the theory .......................................... 13

3 Machine Arithmetic .................................................. 17  
   3.1 Bounded Integer Arithmetic ................................... 18
   3.2 Fixed-Point Arithmetic ....................................... 19
   3.3 Floating-Point Arithmetic .................................... 20

4 A Brief Introduction to Mathematical Logic ........................ 23  
   4.1 Propositional Logic ............................................ 23
   4.2 Predicate Logic ................................................ 25

5 The Satisfiability Problem ............................................ 29  
   5.1 The SAT problem ............................................... 29  
      5.1.1 The DP procedure ........................................ 30
      5.1.2 The DPLL procedure ..................................... 31
      5.1.3 The CDCL algorithm ...................................... 33
   5.2 The SMT Problem ............................................... 36
      5.2.1 Beyond DPLL(T) .......................................... 37

6 Reasoning about Machine Arithmetic ................................ 41  
   6.1 Decision procedures for bit-vectors ........................ 41
      6.1.1 Bit-Blasting ............................................... 41
      6.1.2 Pre-Processing and In-Processing ....................... 42
      6.1.3 Lazy and Eager Approaches .............................. 42
   6.2 Decision procedures for floating-point arithmetic .......... 42
      6.2.1 Encoding into bit-vector arithmetic .................... 42
      6.2.2 The ACDCL approach .................................... 43
7 Contributions 45
8 Conclusion 47
Bibliography 49

Paper I: Approximations for Model Construction

Paper II: Deciding Bit-Vector Formulas with \textsc{mcSAT}
Chapter 1

Introduction

Computer systems permeate modern society, from telecommunications and economy to transportation systems and our households. Considering our reliance on them, the matter of their correctness and safety is of utmost importance. Mistakes can cost enormous sums of money, or worse, human lives. Some examples of these costly errors include:

- the Ariane 5 accident [25]
- the Patriot missile defense bug [5]
- the Pentium FDIV bug [8]
- the Java’s sorting algorithm bug [13]

Preventing such errors requires a huge amount of effort. Computer systems consist of hardware and software working in concert. Testing is the most common approach to detect errors, and it is part of almost every development methodology nowadays. However, for safety-critical applications, testing falls short, since it cannot guarantee absence of errors. Production of both hardware and software is relying more and more on formal methods to ensure that the final product works as intended. They are used, for example, in specification, development and verification. Formal specification removes ambiguities present in natural language and subject it to rigorous examination before providing a formal description of the system. Formal development can use a formal system to synthesize an implementation from the specification. The aim of verification is to show that the implementation is in accordance with the specification. Formal methods require a great deal of expertise and they lend themselves nicely to automation. A goal for the future is to develop techniques and tools that, hopefully, even non-experts can use. One of the main challenges is the increasing size and complexity of
produced software. It makes analysis difficult, if not impossible, despite the automation of these techniques.

Verification is a field of computer science concerned with correct behavior of software. Verification, simply put, asks if the software behaves in accordance with the specification. There are several approaches to answer this question. Some try to answer the question at runtime, like dynamic verification, while static verification methods try to show correctness of software without running it. This approach requires several components: 1. a formal setting to reason in (e.g., first-order logic usually in conjunction with some theory), 2. a program logic (e.g., a Hoare-style calculus) 3. a reasoning procedure for the chosen formal setting.

When discussing correctness of software, we focus mostly on safety-critical software. Imagine an airplane flight controller. It relies on various parameters and arithmetic computation to adjust vector and speed of the aircraft. Ensuring that the controller’s computations are correct and safe (i.e., they satisfy safety properties) depends on our ability to reason about the arithmetic taking place inside. This task in itself quite difficult. However, implementation tricks are often encountered in practice, which can make reasoning about program properties even more difficult. Floating-point arithmetic, as an approximation of the real arithmetic, can exhibit some unintuitive (and unexpected) behavior. Both integer and floating-point arithmetic see frequent use and involve subtle behavior, which makes reasoning about them very relevant.

Reasoning about machine arithmetic is often done using automated tools called SMT solvers. SMT stands for satisfiability modulo theories and represents a class of problems expressed using first-order logic involving some particular theory. For example, theories relevant for reasoning about machine arithmetic are the theory of bit-vectors and the theory of floating point arithmetic. In this thesis, we present methods that improve our ability to reason about these theories.

1.1 Research questions

Work presented in this thesis aims to improve the state-of-the-art in SMT for reasoning about machine arithmetic. Two main research questions are addressed in this thesis:

1. Can use of approximations improve performance and scalability of existing procedures?

2. Is it possible to devise an SMT solver with native support for machine arithmetic?
1.1. Research questions

The first question explores the possibility of pushing the capabilities of existing procedures through use of approximations. In recent years, abstractions and approximations have shown themselves to be a very useful tool to combat the ever increasing problem size. The idea is to abstract away technical details from the problem in order to focus on its core. We investigated a systematic way of applying approximations to the floating-point constraints and present the results in Paper I.

The second question tries to find an alternative to a well established approach to reasoning about machine arithmetic. Namely, many solvers encode machine arithmetic into a less expressive theory, e.g., the theory of floating-point arithmetic into the theory of bit-vectors and the theory of bit-vectors into propositional logic. The advantage is that we can use existing tools to solve this problems. The obvious downside is that the resulting formula can be too large to even begin solving. Another limitation is that structural information is lost in the encoding. For instance, the fact that certain bits are part of the same bit-vector or that a collection of operations is actually an addition operation has no relevance in the propositional world. We investigate a model constructing decision procedure for the theory of bit-vectors, with bit-vectors as first-class citizens, presented in Paper II.
Chapter 2

An example

We start with an example which will illustrate the necessary steps to reason about program behavior. Alg. 1 shows a C function computing the value of the $\sin(x)$ function around zero, using its Maclaurin expansion [1], according to the formula:

$$P_n(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{\sin^{(n)}(0) \cdot x^n}{n!}$$

The polynomial $P_n(x)$ approximates the sine function $\sin(x)$ for the values of $x$ close to zero, i.e.,

$$\sin(x) = \lim_{n \to +\infty} P_n(x)$$

The parameter $n$ defines the degree of approximation, the higher the $n$ the smaller the error. In Alg. 1, the computation of $P_n(x)$ is done iteratively. In every iteration, the next element of the sum is computed by multiplying the previous sum element by $x^2$ and dividing it by $i \cdot (i-1)$. This way the sum elements are computed incrementally, avoiding repeated computations from scratch.

Most of the variables are double precision floating-point numbers, with the exception of the counter variable $i$ which is integer. The higher order derivatives are abstracted away in the code, because they cycle through values $1, 0, -1, 0$. The iterations when the value of the higher-order derivative is zero are simply skipped (by increasing the counter variable $i$ by two). In the remaining cases an if-then-else statement alternates the sign. The stopping condition for the loop is when a predefined precision $\text{EPS}$ is achieved.

When considering code in Fig. 1, we might wonder if some simple properties of the sine function are preserved. For example, the sine function is limited by 1 and -1, $|\sin(x)| \leq 1$ for all $x$. It is also symmetric around zero,
#include <math.h>

#define EPS 1E-20

double sine_expansion(double x){
    double x_sq = x * x;
    double ratio = x;
    double sum = ratio;
    int i;
    for (i = 3; abs(ratio) > EPS; i+=2) {
        ratio *= x_sq/(i*(i-1));
        if (i%4 == 1)
            sum += ratio;
        else
            sum -= ratio;
    }
    return sum;
}

Algorithm 1: Maclaurin series expansion of the \( \sin(x) \) function.

\[ \sin(x) = \sin(-x) \] for all \( x \). We can show these properties mathematically for the polynomial \( P_n \). The real question, however, is whether the implementation \( \text{sine
d_expansion} \) satisfies these properties. Is \( |\text{sine_expansion}(x)| \leq 1 \) for all values of \( x \)? And is \( \text{sine_expansion}(x) = \text{sine_expansion}(-x) \) for all \( x \)? In order to discover answers to these questions, we will need to analyze the program execution in some way.

## 2.1 Analyzing the execution

In terms of formal methods there are several ways to approach analysis of program executions. Two categories of interest to us are *deductive* and *model-checking-based* methods. In both cases we have a specification and properties that should be checked. Deductive methods rely on mathematical proof systems to show that properties are satisfied. Given, say, an axiomatic system and its rules, a proof is constructed by repeated rule application, until proof obligations are discharged. The proof can be produced using, e.g. interactive theorem provers, automated theorem provers or SMT solvers. In general, these methods allow the user to provide some crucial pieces
of information to help the prover conclude the proof (e.g., by supplying a critical statement or a lemma). Model-checking-based approaches try to find a description of a system (usually in the form of a finite state transition system) that will be a logical model of the desired properties. The property is usually given as a formula in some temporal logic, such as CTL [7]. Proving that the system is a model of the formula is done by systematic exploration of the search space and explicit test in each state for the desired property. Model-checking-based methods have the advantage of being fully automatic, but suffer from scalability issues (also known as the state space explosion problem). These methods include model checking [7] and bounded model checking [3].

We apply bounded model checking to the program in Fig. 1. Given a program and a property, we want to check if all the executions of the program satisfy the property. A program is interpreted as a state transition system, where each state is represented by the current values of program variables. Executing a command in a given state will result in changes to the variables (and therefore the state). The transition system is encoded into a formula $\psi_{\text{program}}$, which describes semantics of the program. This is straightforward for programs that contain no loops or recursion. Handling loops requires unrolling the loop a fixed number of times, say $n$, and then constructing the formula. By fixing the number of loop iterations we restrict ourselves to a particular class of program executions against which we will check the safety property. Let formula $\psi_{\text{property}}$ encode the property that we want to check. We will check whether the property is satisfied for all executions, by asking whether there are any that violate it. We will then ask an SMT solver whether

$$\psi_{\text{program}} \land \neg \psi_{\text{property}}$$

is satisfiable. If it is satisfiable, that means that for some combination of inputs, the program can reach a state (within $n$ loop iterations) where the desired property does not hold. In that case, the model of the formula can be used to reconstruct an execution trace that violates the property $\psi_{\text{property}}$. If there is no model, then we can conclude that there are no executions of length $n$ that violate the property. In general, BMC cannot show that the property is truly satisfied for all executions, but it will discover any executions of the chosen length $n$ that violate the property.

When considering how to encode program executions as formulas, two things attract attention. The first thing is that programs are sequential (i.e., statement order is important), whereas formulas have no such considerations. The second is that variables in programs can hold different values during an execution, while mathematical variables are constant. To capture these two properties in the world of first-order logic, we introduce a
sequence of mathematical variables for each program variable. These mathematical variables will have subscripts corresponding to their occurrence in the program, e.g., a program variable \( i \) will be associated with \( i_{\text{init}}, i_0, i_1, \ldots \). Every time an assignment to a variable occurs, we will use a variable indexed with a new label. This corresponds to the single static assignment form (SSA) \([10]\), which requires that each variable is defined and assigned only once during the program execution. Since the example contains only initialization and the loop, we will label the variables with \( \text{in} \) for input argument values, \( \text{init} \) for values at time of initialization, and with the current value of the counter variable \( i \) when the assignment occurs in all other cases. Variable occurrences on the left-hand side of the assignment statement always use a new label, while occurrences of variables on the right-hand side use the last introduced label for that variable up to that point.

**Example 2.1.1.** The simplest execution to consider is one where the loop is not executed at all. In that case, the initialization of variables is the only thing that will occur.

\[
\begin{align*}
\text{x\_sq_{init}} &= \text{x\_{in}} \ast \text{x\_{in}}; \\
\text{ratio\_{init}} &= \text{x\_{in}}; \\
\text{sum\_{init}} &= \text{ratio\_{init}}; \\
\text{i\_{init}} &= 0; \\
\text{i\_0} &= 3; \\
\{!(\text{fabs}(\text{ratio\_{init}}) > \text{EPS})\} & \Rightarrow \text{return \ sum\_{init}};
\end{align*}
\]

Figure 2.1: This figure shows: (a) the execution of Alg. 1 with no loop unrollings and (b) the corresponding first-order formula \( \phi_0(x_{\text{in}}) \).

Let’s consider the control flow of execution in Fig. 2.1. Since the body of the loop is not executed, the loop condition \( \text{fabs}(\text{ratio}) > \text{EPS} \) is not satisfied. That will be the case whenever the input value \( x_{\text{in}} \) is smaller or equal than the predefined value \( \text{EPS} \).

We will proceed to construct a series of formulas \( \phi_i(x_{\text{in}}) \) where \( i \) denotes number of loop unrollings. The formula with no unrollings, denoted \( \phi_0(x_{\text{in}}) \), is shown in Fig. 2.1b. It is simple to construct, because each variable is assigned only once and nothing particularly exciting is happening.

Fig. 2.2 shows an execution of the program where the loop body is executed exactly once. It models all executions where the initial condition of the for-loop was met, but was violated after only one execution of its body. Usually, the goal of this technique is to find traces (executions) which
2.1. Analyzing the execution

```
x_sq_init = x_in * x_in;
ratio_init = x_in;
sum_init = ratio_init;
i_init = 0;
i_0 = 3;
assume(fabs(ratio_init) > EPS)
ratio_3 = ratio_init * (x_sq_init / (i_0 * (i_0 - 1)))
if (i_0 % 4 == 1) then
    sum_3 = sum_0 + ratio_3;
else
    sum_3 = sum_0 - ratio_3;
i_3 = i_0 + 2;
assume(!(fabs(ratio_3) > EPS))
return sum_3;
```

Figure 2.2: Execution of the program in Fig. 1 with one loop iteration being executed.

violate the desired property. If the property is monotonic in the sense that once violated it will stay violated we can just generate one long execution and then check whether the property holds. Otherwise, we generate the executions each time unwinding one more loop iteration. We can now continue unwinding the loop for various values of \(n\) to obtain sequences of executions to analyze.

By unwinding the loop, we arrive at a shape of program that is easy to analyze. We will need to take into account the additional conditions, such as, the met or failed loop conditions. These can now be easily translated into a formula that describes exactly the control flow during the program execution. To differentiate between variable types and their corresponding operations, the encoding is done in a multi-sorted first-order logic with support for the involved theories. For the purposes of illustration, we will write the formula \(\phi_0(x)\) in the SMT-LIB format [2], which is the standard input language for SMT solvers. It will rely on two theories of the SMT-LIB, the quantifier-free theory of floating-point arithmetic and the quantifier-free theory of fixed-length bit-vector arithmetic.

**Example 2.1.2.** Given a program execution in SSA form (each variable is assigned exactly once), we produce the corresponding SMT-LIB formula, shown in Fig. 2.3. An SMT-LIB file consists of several parts: 1. preamble — which can declare meta information, such as the theory being used, author of the file, comments, status of the formula, etc. 2. function declarations —
(set-logic QF_FP)

(declare-fun x_in () (_ FloatingPoint 11 53))
(declare-fun x_sq_init () (_ FloatingPoint 11 53))
(declare-fun ratio_init () (_ FloatingPoint 11 53))
(declare-fun sum_init () (_ FloatingPoint 11 53))
(declare-fun i_init () (_ BitVec 32))
(declare-fun i0 () (_ BitVec 32))

(assert (= x_sq_init (fp.mul x_in x_in)))
(assert (= ratio_init x_in ))
(assert (= sum_init ratio_init))
(assert (= i_init (_ bv0 32)))
(assert (= i0 (_ bv3 32)))
(assert (not (fp.gt (fp.abs ratio_init) (fp (_ bv0 1) (_ bv989 11)
                                          (_ bv3233525618163131 53)))))

(assert (not (fp.leq (fp.abs sum_init) (fp (_ bv0 1) (_ bv1023 11) (_ bv0 53)))))

(check-sat)

Figure 2.3: $\phi_0(x)$ in SMT-LIB format

denoted by keyword declare-fun followed by the function name, sorts of
its arguments (empty brackets denote zero arguments, i.e., function symbol
is a constant) and finally the sort of the return value, 3. the formula itself — a list of assertions that should hold, 4. call to check for satisfiability — in the form of check-sat keyword.

In Fig. 2.3, the SMT-LIB keywords are written in bold, while the theory
specific keywords, such as sorts, constructors, constants and operations are
written in blue color. When analyzing a program, we have to choose the
reasoning domain. Since the SMT-LIB has support for the theory of floating-
point arithmetic, we will use it to encode the program execution.

On line 1, the logic (i.e., the theory) of the formula is declared. In this
case, it is the quantifier-free logic of floating-point arithmetic (or QF_FP).
It contains the FloatingPoint and RoundingMode sorts, constants for the
five rounding modes and all the standard floating-point operations. Nor-
mally, the simple arithmetic operations such as $+, -, \cdot, \ldots$ are overloaded in
programming languages. The SMT-LIB logics disambiguate between them, by prefixing each operation with a corresponding acronym for the theory. For example, the floating-point operations $\oplus, \odot, \oslash$ are denoted as fp.add, fp.sub, fp.mul in $\text{QF}_\text{FP}$. The lines 3-9 declare the constants corresponding to the program variables and their sorts. For example, on line 3, $x_{\text{in}}$ is declared as a mathematical constant with FloatingPoint sort with $11$ exponent bits and $53$ significand bits (i.e., a double precision floating-point number). Integer values have the BitVector sort and concrete values are specified using a constructor. For example, $(\_ \text{ bv3 } 32)$ denotes a bit-vector literal of bit-width $32$ representing integer value $3$. These lines correspond to program declarations (and in some cases initializations). Lines 10-16 describe the semantics of the 0-unfolding program execution. Each assignment is treated as an equality between the variable and the expression being assigned. Such equalities must hold if the formula is to encode the program behavior, so they are asserted as true (using the assert keyword). Next, the negated property, $\neg (\text{sum} \leq 1)$, is asserted on line 20. Finally check-sat instructs the solver to check the satisfiability of the formula.

2.2 Choosing the theory

In the previous example we chose to encode the program as a first-order formula of the theory of floating-point arithmetic. This is a natural approach since we want to take into account the intricacies of the FPA semantics as described by the IEEE-754 standard. However, it is not the only approach. We could take the same execution and encode the floating-point operations as operations over reals, fixed-point arithmetic or even integers. One might want to reason about the program involving reals because that is what the program is supposed to compute, whereas floating-point numbers are merely a vehicle of computation. These are different problems, each one with its own flavor of semantics. We can reason about the real arithmetic, the the IEEE-754 standard-compliant floating-point arithmetic, the fixed point arithmetic, or the bounded integer arithmetic. Each of these domains and their semantics we will refer to as a theory. For these theories there exist algorithms, called decision procedures, which can determine whether a formula is a theorem of that particular theory. We will briefly introduce some of the decision procedures for these theories.

- SMT-style decision procedures for floating-point arithmetic can be roughly divided into two categories: eager and lazy. The eager approach looks at the the IEEE-754 standard-compliant hardware and see how floating-point arithmetic is carried out in the registers and
encodes it into bit-vector arithmetic which is then reduce to propositional satisfiability (SAT) by a method called bit-blasting [22]. Then we can check satisfiability using a SAT solver. This method can also easily provide a model for the original formula. However, it suffers from scalability issues, because it combines two encodings (from FPA to BVA and from BVA to SAT) which results in very large formulas. An example of the lazy approach the ACDCL algorithm [6]. Instead of explicitly representing the floating-point values, they can be abstracted as intervals (i.e., pairs of bounds \((\text{lower, upper})\)). The ACDCL algorithm reasons about the interval lattice and is capable of conflict analysis and learning while respecting the semantics of FPA. This approach excels at determining unsatisfiability, with the downside that the model might take same time to obtain even when satisfiability becomes obvious. Beside the SMT approaches, there exist constraint solvers which implement standard-compliant semantics [27]. Semi-automated approaches using interactive theorem provers also exist. For example, the Coq proof assistant [16] can be used to formally reason about floating-point arithmetic [26].

- There are several decision procedures for the real arithmetic. For linear real arithmetic, the simplex method and the Fourier-Motzkin procedure are the most prominent [22]. There are also procedures for non-linear arithmetic, but they are not very efficient. In general, we can use efficient linear programming solvers, which are very good at solving linear constraints, but would have little effect on non-linear problems.

- Fixed-point arithmetic is considerably simpler way to represent real values than floating-point arithmetic. It is less expressive, but also easier to reason about. It can be encoded into theory of bit-vectors in a straightforward manner, since its essentially a non-standard interpretation of the bounded integer arithmetic.

While the theories mentioned above can all be used to model real values in programs, they all have their own particular semantics. A very natural question arises. Is possible to use a decision procedure of one theory to reason about another theory. For example, could we use reals to reason about floating-point arithmetic? And what is there to be gained by it? The answer is yes, it is possible to reason about floats using reals, but it comes with a caveat. Since they have different semantics, solutions over reals might exist that simply cannot be represented using floating-point numbers. Conversely, there could be floating-point solutions that are overlooked in the theory of reals. A simple example is the associativity property, which holds
over reals, but does not hold in floating-point arithmetic due to rounding. In this thesis we show how cross-theory reasoning can performed, in a way that guarantees that found solutions are indeed solutions and that no solutions are lost.
Chapter 3

Machine Arithmetic

Most number systems used in computers are positional. That means that each number is stored as a string of digits, and that position of a digit determines its contribution to the overall value. The set of digits available depends on the the base $b$ of the number system, also called the radix. The digits then belong to the set \{0, \ldots, b - 1\}. Base 10, or the decimal numeral system, is the commonly taught system and one that most people are familiar with. When it comes to computers, the natural choice for the base is $b = 2$, also called the binary numeral system. Historically bases 3, 8, 10 and 16 have seen use, and some of them (8, 10 and 16) are used when convenient. For example, octal (base 8) numeral system can be obtained from the binary by grouping three adjacent binary digits and mapping them to an octal digit. Similarly, hexadecimal (base 16) representation is obtained by grouping 4 adjacent binary digits. While their representation is not explicitly octal or hexadecimal, they enable more compact representation of binary literals that would be too long to write down elegantly. The number $d$ (or its value) denoted by $(d_n d_{n-1} \ldots d_1 d_0. d_{-1} \ldots d_{-p})_b$ in a positional system with base $b$ is calculated as:

$$d = \sum_{i=-p}^{n} d_i \cdot b^i$$  \hspace{1cm} (3.1)

We use numerical systems to represent different sets of numbers. Integers and reals are of interest to this thesis, so we will look at them in greater detail. Representing integer values does not use digits beyond the decimal point, i.e. $p = 0$. Representing real values makes use of all the digits and depending on the values the notation shown above can be cumbersome (e.g., extremely small values will have many leading zeroes after the decimal point). Both representations have difficulties representing large values succinctly, especially if all we care about is the order of magnitude (since the least valued bits will be practically negligible).
The scientific notation is an elegant way of representing numbers that might be too great or too small to represent conventionally. In scientific notation, a number is written down as a product of its significant digits and a power of the number system base:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

This representation is not unique, but a normal form can be imposed upon it. For example, requiring that $0 < |\text{significand}| < \text{base}$ yields a normal form. So does the requirement that the exponent has a particular value and there are others still. The basic notation shown in 3.1 corresponds to $\text{exponent} = 0$, but other values could be used if we are dealing with a particular range of small or large values. In the case of zero, the base and exponent are omitted altogether. The exponent easily conveys the order of magnitude of a number, while still presenting its most important significant digits, allowing for easy comparisons.

Representing numerical data in a computer faces some (mathematically) unexpected behaviour. This is usually the consequence of the fact that we do not have infinite memory at our disposal, so we have limit ourselves to a certain size in memory. The most obvious consequences are overflows and rounding error, but we will see that there are also some more subtler effects that occur.

Representing integral and real data in computers can be done using different possible data types, depending on the hardware and the programming language. Integers are used when dealing with integral data, while fixed-point and floating-point numbers are used to represent real-valued data. Fixed-point arithmetic is used when a floating-point unit is unavailable (e.g., some micro controllers) or when we know exact magnitude (and precision) necessary for the represented values. Floating-point numbers are more complex and allow a greater magnitude of values at the expense of precision. The two representations differ in the density of representable data points. Fixed-point arithmetic has uniform density, while floating-point arithmetic has variable density (and error).

### 3.1 Bounded Integer Arithmetic

Integral data is usually associated with the fixed-size integer data type. Depending on the system, programming language and implementation, we usually distinguish between short, integer, long and long long by their bit-width. They can also be recorded as signed and unsigned integers, which affect the range of values that can be represented. In terms of the scientific notation, the exponent is fixed to 0, base to 2 and significand has
3.2. Fixed-Point Arithmetic

Fixed-point arithmetic is an equivalent of scientific notation where the exponent is set to have a predetermined value $N$, also called the precision. It determines the number of fractional digits after the radix point and is chosen carefully based on the data to be represented. One can view fixed-point arithmetic as integer arithmetic where the unit of measure is smaller than 1. It has a uniform density of real values that can be represented. Typically, fixed-point arithmetic implemented using integers as the underlying data type and the operations as macros (or inlined functions) for performance reasons, mainly to avoid frequent function calls on the stack. Due to fixed length, there is a trade off between precision and magnitude of the values that can be represented. Every fractional digit means one less digit.
for the integral part of the value. Some fixed-point operations, like addition and subtraction, can be implemented using the corresponding integer operations. Some others, like multiplication and division, produce results that will not be aligned with the chosen radix position. These operations require some shifting to align the results with representation. Additionally, they are vulnerable to precision loss, mainly due to overflows of intermediate integer operations. These can be avoided, but require some additional effort. When a value has more fractional digits than the chosen precision, rounding takes place. Rounding is usually towards zero, but other rounding modes are possible.

3.3 Floating-Point Arithmetic

Intuitively, floating-point arithmetic, corresponds to a scientific notation with the requirement $0 \leq |\text{significand}| < base$. Additionally, the significand and the exponent are subjected to a limited number of digits. However, floating-point arithmetic includes several special values: signed zeros, signed infinities and special not-a-number values. In the remainder this section, we will see in more detail how floating-point arithmetic is defined.

Floating-point arithmetic is defined in the IEEE-754 standard [21]. It is a systematic approximation of real arithmetic using a finite representation. The approximation involves several layers of specification, moving from the domain of reals to a domain of bit-strings represented in memory.

The highest specification level is the domain of extended reals, which includes special values for infinities $+\infty$ and $-\infty$. This infinite set is mapped, using rounding, to a finite set of floating-point values. This set is algebraically closed and includes special values for signed zero, $-0$ and $+0$, and not-a-number value $NaN$. The set of floating-point values is then mapped to a particular representation of floating-point data. The representation of floating-point data includes the sign-exponent-significand triples and all the special values (now distinguishing between quiet and signaling NaNs). The fourth and final level is that of actual bit-string encodings that are stored and manipulated. Some floating-point data have multiple encodings, such as the NaNs, which store diagnostic information as part of its encoding. The standard specifies several formats, three binary and two decimal ones, with encodings of various lengths. The standard allows for various sizes of bit-vectors that are used to represent the significand and the exponent of numbers; e.g., double-precision floating-point numbers are represented by using 11 bits for the exponent and 53 bits for the significand.

We denote the set of floating-point data that can be represented as
3.3. Floating-Point Arithmetic 21

floating-point numbers with $s$ significand bits and $e$ exponent bits by $FP_{s,e}$:

$$
\left\{ (-1)^{sgn} \cdot sig \cdot 2^{exp-s} \mid sgn \in \{0, 1\}, \\
\quad sig \in \{0, \ldots, 2^s - 1\}, \\
\quad exp \in \{-2^{e-1} + 3, \ldots, 2^{e-1}\} \right\} \cup \{ NaN, +\infty, -\infty, -0 \}
$$

The set consists of: 1. normalized numbers (in practice encoded with an implicit leading bit set to 1), 2. subnormal numbers, and 3. special values. The definition does not discriminate between normal and subnormal numbers and any value with multiple representations loses the multiplicity in the set. Since the reals do not contain a signed zero value it is included explicitly with the other special values.

The semantics of floating-point operations defined by the standard is the same as that of operations performed over reals, except that in the case that the obtained value is not representable in the given format then rounding takes place in accordance with the rounding mode. The rounding modes described by the Standard are RoundTowardZero, RoundNearestTiesToEven, RoundNearestTiesToAway, RoundTowardPositive, RoundTowardNegative. Let $\cdot \in \{+,-,\times,\ldots\}$ and $\cdot_{FP}$ denote its counter-part in the theory of floating-point arithmetic. Then an operation using rounding mode $r$ over arguments $a$ and $b$ (in prefix notation) is defined as:

$$(\cdot_{FP} r a b) = (\text{round}(r, (\cdot a b)))$$

Since it is designed as an approximation of real arithmetic, FPA’s use of rounding, introduces errors which would not occur in real arithmetic. The use of rounding also affects some basic mathematical properties of operations, such as associativity which does not hold in FPA.

Floating-point numbers have a dynamic range and density. For each value of the exponent (i.e., magnitude of the number) the significand can represent the same number of different values. As a consequence, the representable values are denser in lower magnitudes, and sparser in greater magnitudes. Varying density combined with rounding leads to varying magnitude of errors during computation, depending on the values involved. Error of operations is then expressed in units-in-last-place, or ulp. Another peculiar interactions is that, addition of a non-zero floating-point value to a value of much greater magnitude can appear as a zero-addition. A big difference in magnitude of values can affect stability of numerical algorithms. Some techniques from numerical mathematics, such as pre-sorting of values, can help offset the problem, but at an additional computation cost. Using FPA can be tricky. Special values usually propagate through computations. If the programmer loses sight of this fact and produces code that ignores emergence of special values, the entire computation could be compromised.
Chapter 4

A Brief Introduction to Mathematical Logic

In this section, we give a short introduction to basic elements of propositional and predicate logic. Both of them serve as a rigorous formal language used to precisely describe problems from many areas of computer science. Propositional logic is well suited to reasoning about finite domains, while predicate logic can reason about infinite domains. For each of them, we will first present the syntax, the grammar of the logic and then follow up with their semantics (that is, the meaning of the formulas).

4.1 Propositional Logic

Propositional logic uses variables to represent propositions (or truth statements), which can be combined into more complex propositions.

Syntax The language of propositional logic, i.e., the way to form syntactically correct propositional formulas, is defined by the syntax.

The alphabet of propositional logic $\Sigma$ consists of:

- constant literals: $\top$ (true) and $\bot$ (false),
- a countable set of propositional variables $P$,
- logical connectives: $\neg$ (negation), $\land$ (conjunction), $\lor$ (disjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence)
- auxiliary symbols: '(' and ')'

The set of all well-formed propositional formulas over a set of variables $P$ is the smallest subset of the set of all words of the alphabet $\Sigma$ such that:
• logic constants are propositional formulas
• propositional variables are propositional formulas
• if $A$ and $B$ are propositional formulas then so are: $(\neg A)$, $(A \land B)$, 
  $(A \lor B)$, $(A \Rightarrow B)$, $(A \leftrightarrow B)$.

Constant literals and propositional variables are called atomic formulas. A literal is an atomic formula or its negation. A disjunction of literals is called a clause, while a conjunction of literals is called a cube.

A formula is said to be in negation normal form, if it contains only $\neg$, $\land$, $\lor$ connectives and $\neg$ is applied only to variables.

A formula that is a conjunction of clauses is said to be in conjunctive normal form, i.e., if it has the following shape:

$$\bigwedge_{i \in I} D_i,$$

where $D_i$ is a clause, for all $i \in I$. A formula that is a disjunction of cubes is said to be in disjunctive normal form, i.e., if it has the following shape:

$$\bigvee_{i \in I} C_i,$$

where $C_i$ is a cube, for all $i \in I$.

Semantics The meaning of formulas is defined by the semantics. Functions $v : P \mapsto \{0, 1\}$ are called valuations, and the set $\{0, 1\}$ the valuation domain. Every valuation defines a recursive evaluation function $val_v(\cdot)$, which maps every formula to the set $\{0, 1\}$. The valuation function is defined in the following manner:

• $val_v(\top) = 1$ and $val_v(\bot) = 0$

• $val_v(x) = v(p)$, $p \in P$

• $val_v(\neg A) = \begin{cases} 
  1, & \text{if } val_v(A) = 0 \\
  0, & \text{if } val_v(A) = 1 
\end{cases}$

• $val_v(A \land B) = \begin{cases} 
  1, & \text{if } val_v(A) = 1 \text{ and } val_v(B) = 1 \\
  0, & \text{otherwise} 
\end{cases}$

• $val_v(A \lor B) = \begin{cases} 
  0, & \text{if } val_v(A) = 0 \text{ and } val_v(B) = 0 \\
  1, & \text{otherwise} 
\end{cases}$
4.2. Predicate Logic

\[ val_v(\phi \implies B) = \begin{cases} 0, & \text{if } val_v(A) = 1 \text{ and } val_v(B) = 0 \\ 1, & \text{otherwise} \end{cases} \]

\[ val_v(A \iff B) = \begin{cases} 1, & \text{if } val_v(A) = val_v(B) \\ 0, & \text{otherwise} \end{cases} \]

We say that a formula \( A \) is true under valuation \( v \) if \( val_v(A) = 1 \), and the valuation \( v \) is said to be satisfying the formula \( A \). If \( val_v(A) = 0 \) we say that the formula \( A \) is false under the valuation \( v \).

We say that a formula is satisfiable if there exists a satisfying valuation for it. A formula is called a tautology if every valuation is a satisfying valuation. A formula is a contradiction (or unsatisfiable) if a satisfying valuation does not exist.

**Example 4.1.1.** Formula \( p \implies p \) is a tautology, \( p \land \neg p \) is a contradiction and \( p \implies q \) is satisfiable.

4.2 Predicate Logic

*Predicate logic* or *first-order logic* is more expressive than propositional logic. It allows reasoning about non-Boolean domains and, even more, reasoning about infinite domains through the use of quantifiers. We establish a formal basis in the context of multi-sorted first-order logic (e.g., [19]).

**Syntax** First-order language consists of the logical part (which mostly coincides with that of propositional logic) and the non-logical part (also called the *signature*).

The logical part of first-order language consists of:

- constant literals \( \top \) and \( \bot \),
- a countably infinite set of variables \( X \),
- basic logical connectives \( \neg, \land, \lor, \Rightarrow, \iff \)
- quantifiers \( \forall \) (*universal*) and \( \exists \) (*existential*)
- auxiliary symbols: ‘(’ and ‘)’

A first-order signature \( \Sigma = (S, P, F, \alpha) \) consists of:

- a set of sort symbols \( S \),
- a set of predicate symbols \( P \),
• a set of function symbols $F$,
• and a sort mapping $\alpha$.

Every predicate symbol $p \in P$ is assigned a $k$-tuple $\alpha(p)$ of argument sorts, where $k$ is the arity of $p$. Each function symbol $f \in F$ is assigned a $(k + 1)$-tuple $\alpha(g)$ with $k$ argument sorts (with $k \geq 0$) and the sort of the result. Again, $k$ is the arity of the symbol. Nullary function symbols are called constants. We overload $\alpha$ to assign sorts also to variables.

Given a multi-sorted signature $\Sigma$ and variables $X$, we define the notions of well-sorted terms, atoms, literals, clauses. Variables and function symbols of arity 0 are called terms. Terms are also applications of $k$-ary function symbol $f$ to terms $t_1, t_2, \ldots, t_k$, where the sorts of terms $t_i, i \in \{1, 2, \ldots, n\}$ match the sorts of arguments of the function symbol $f$.

Application of a $k$-ary predicate symbol $p$ to terms $t_1, t_2, \ldots, t_k$, where the sorts of terms $t_i, i \in \{1, 2, \ldots, n\}$ match the sorts of arguments of the function symbol $f$, is called an atomic formula. A literal is an atomic formula or its negation.

We say that atomic formulas are formulas. If $\phi$ and $\psi$ are formulas then so are $\neg \phi$, $\phi \land \psi$, $\phi \lor \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$. If $x$ is a variable and $\phi$ a formula then, $\forall x. \phi$ and $\exists x. \phi$ are also formulas.

**Semantics** A $\Sigma$-structure $m = (U, I)$ with underlying universe $U$ and interpretation function $I$ maps each sort symbol $s \in S$ to a non-empty set $I(s) \subseteq U$, each predicate symbol $p \in P$ of sort $\alpha(p) = (s_1, s_2, \ldots, s_k)$ to a relation $I(p) \subseteq I(s_1) \times I(s_2) \times \ldots \times I(s_k)$, and each function symbol $f \in F$ of sort $\alpha(f) = (s_1, s_2, \ldots, s_k, s_{k+1})$ to a set-theoretic function $I(f) : I(s_1) \times I(s_2) \times \ldots \times I(s_k) \rightarrow I(s_{k+1})$. A variable assignment $\beta$ under a $\Sigma$-structure $m$ maps each variable $x \in X$ to an element $\beta(x) \in I(\alpha(x))$.

The recursive evaluation function $val_{m, \beta}($) is defined for terms and formulas. We omit the cases for logical constants and basic logic connectives, because they remain unchanged compared to propositional logic.

• $val_{m, \beta}(x) = \beta(x)$, where $x \in X$
• $val_{m, \beta}(f(t_1, t_2, \ldots, t_n)) = I(f)(d_1, d_2, \ldots, d_n)$, where $f \in F$ and $d_i = val_{m, \beta}(t_i)$, for all $i \in \{1, 2, \ldots, n\}$
• $val_{m, \beta}(\exists x. \phi) = 1$ if there exists an assignment $\beta'$ differing from $\beta$ at most in the value of $x$, such that $val_{m, \beta'}(\phi) = 1$. Otherwise, $val_{m, \beta}(\exists x. \phi) = 0$.
• $val_{m, \beta}(\forall x. \phi) = 1$ if no assignment assignment $\beta'$ exists, such that $\beta'$ differs from $\beta$ at most in the value of $x$ and $val_{m, \beta'}(\phi) = 0$. Otherwise, $val_{m, \beta}(\forall x. \phi) = 1$. 
A theory $T$ is a pair $(\Sigma, M)$ of a multi-sorted signature $\Sigma$ and a class of $\Sigma$-structures $M$. For example, real arithmetic and floating-point arithmetic are theories in this sense. A formula $\phi$ is $T$-satisfiable if there is a structure $m \in M$ and a variable assignment $\beta$ such that $\phi$ evaluates to true; we denote this by $m, \beta \models_T \phi$, and call $(m, \beta)$ a $T$-solution of $\phi$. If formula $\phi$ is said to be valid if it evaluates to true in every $\Sigma$-structure and for every assignment $\beta$. 


Chapter 5

The Satisfiability Problem

The problem of determining whether there exists a satisfying assignment for a propositional formula is called the *satisfiability problem* or the SAT problem. The SAT problem lies at the very heart of computer science, as numerous theoretical and practical problems depend on our ability to determine whether a set of constraints can be satisfied. The SAT problem is computationally hard, in the sense that an efficient (i.e., polynomial complexity) algorithm is unknown. Moreover, the SAT problem is the first problem to be proven NP-complete, by Cook [9]. Levin independently discovered the same result [24], and the result is nowadays known as the Cook-Levin theorem.

In recent years, an more expressive alternative to encodings into SAT has emerged in the form of *satisfiability modulo theories* (SMT). SMT is the problem of checking the satisfiability of a first-order formula in the presence of background theories. Reasoning about machine arithmetic relies on both SAT and SMT solvers. Therefore we briefly delve into history and insights of algorithms behind the SAT and SMT technology.

5.1 The SAT problem

Besides its great importance to theoretical computer science, the SAT problem has many practical applications, mainly due to many efficient implementations in the form of SAT solvers. They also play an integral part of the SMT technology. Historically, the DP [12] and its extension, the DPLL [11] procedures are the first, named after its authors Davis, Putnam, later joined by Logeman and Loveland. Its modern version is called *conflict driven clause learning* algorithm, and it leverages multiple insights and highly efficient data structures to improve upon the original algorithms. Other approaches exist, such as stochastic and non-CNF solvers. In this thesis the focus is on the DPLL-based algorithms due to their relevance to
the SMT technology.

Input for the SAT problem is usually a formula in CNF. The algorithms output SAT if there exists a variable assignment that satisfies the formula, and UNSAT if there is no variable assignment that satisfies the formula. A CNF formula $F$ is represented as a set of clauses which should all be satisfied simultaneously. Clauses are sets of literals of which at least one should be assigned value $true$. An empty set of clauses is considered satisfiable. A formula containing an empty clause is considered to be unsatisfiable. If a clause $c$ consists of only one literal $l$, then we call $c$ a unit clause, denoted by $(l)$. An empty clause is denoted by $(\ )$. A literal $l$ is called pure if its negation does not occur in $F$. Given a formula $F$ in CNF and a literal $l$, with $F[l \mapsto \top]$ we denote a formula in which each occurrence of $l$ is replaced by $\top$ and each occurrence of $\neg l$ is replaced by $\bot$. In practice, we immediately simplify the formula. Every clause that contains $\top$ is deleted, because it is already satisfied. Similarly every occurrence of $\bot$ is deleted, because it does not satisfy the clause in which it occurs.

### 5.1.1 The DP procedure

The DP procedure for propositional satisfiability is named after the authors Martin Davis and Hillary Putnam [12]. It was proposed as part of an algorithm for checking satisfiability of first-order formulas using the Herbrand theorem, by enumerating ground instances of the formula until an unsatisfiable set of ground instances is found. The DP procedure is shown in Alg. 2.

The algorithm applies the following set of rules:

1. An empty set of clauses is considered satisfied and it returns SAT.
2. If the formula contains an empty clause, it is unsatisfiable and the returns UNSAT.
3. unit propagation — If there is a unit clause $c = (l)$, then the literal $l$ has to be true. All clauses that contain $l$ are removed, and all occurrences of $\neg l$ are deleted from the remaining clauses.
4. pure literal — A pure literal $l$ is set to true and all clauses that contain $l$ are removed from $F$.
5. variable elimination — If none of the above rules can be applied, then a variable needs to be eliminated and a new formula constructed. This is done by factoring out all the positive and negative occurrences of the variable. Factoring out all occurrences of variable $p$ in $F$ yields an equivalent formula $F'$ of the form: $(A \lor p) \land (B \lor \neg p) \land R$, where $A$,
5.1. The SAT problem

$B$ and $R$ contain no occurrences of $p$ and $¬p$. A crucial observation is that formula $F'$ is satisfiable if and only if the formula $(A ∨ B) ∧ R$ is satisfiable. The procedure then proceeds to solve formula $(A ∨ B) ∧ R$ instead of $F'$. Removing the variable in this way creates a smaller problem (in number of variables), but usually with a greater number of clauses.

**Input**: Formula $F$ in CNF  
**Output**: SAT if $F$ is satisfiable, UNSAT otherwise

```
begin
    if $F$ is empty then
        return SAT
    end
    if $(l) ∈ F$ then
        return UNSAT
    end
    if $(l) ∈ F$ then
        return DP ($F[l ← T]$)
    end
    if $l$ is pure in $F$ then
        return DP ($F[l ← T]$)
    end
    return DP (eliminate_variable ($F$))
end
```

**Algorithm 2**: The DP procedure

5.1.2 The DPLL procedure

The Davis-Putnam-Logeman-Loveland (DPLL) procedure [11] builds upon the DP procedure by introducing the *split rule*. Instead of eliminating a variable from the formula, case split is done instead. Consider again the formula $F' = (A ∨ p) ∧ (B ∨ ¬p) ∧ R$ (which is equivalent to $F$). $F'$ is unsatisfiable if and only if both $F'[p ← T] = B ∧ R$ and $F'[-p ← T] = A ∧ R$ are unsatisfiable. The procedure first attempts to solve the problem with literal $p$ set to true, i.e., $F[p ← T]$. If that fails, then it solves $F$ with literal $p$ set to false, i.e., $F[¬p ← T]$. The advantage of the split rule is that it avoids generation of great number of additional clauses, many of which grow long and redundant over time. By making the variable assignments explicit, the search space gains a tree structure. The DPLL procedure is shown in Alg. 3.
Input : Formula $F$ in CNF
Output: SAT if $F$ is satisfiable, UNSAT otherwise

begin
  if $F$ is empty then
    return SAT
  end

  if () $\in F$ then
    return UNSAT
  end

  if ($l$) $\in F$ then
    return DPLL ($F[l \mapsto T]$)
  end

  if $l$ is pure in $F$ then
    return DPLL ($F[l \mapsto T]$)
  end

  Choose a variable $p$ in $F$
  if DPLL ($F[p \mapsto T]$) = SAT then
    return SAT
  else
    return DPLL ($F[\neg p \mapsto T]$)
  end
end

Algorithm 3: The DPLL procedure
Abstract Transition System for the DPLL algorithm

The performance of DPLL algorithm is dependent on the choices of variables taken during the search. Different heuristics can be used to choose the right variable. However, these aspects are not interesting from perspective of the core algorithm. To abstract away such details away, the DPLL algorithm can be presented as an abstract transition system [28]. The rules applied in Alg. 3 are presented as transitions of the system. We distinguish between the search states \( \langle M, C \rangle \), and two accepting states \( \text{unsat} \) and \( \text{sat} \). A search state \( \langle M, C \rangle \) consist of the partial model \( M \) called the trail and the (multi)set \( C \) of clauses to be satisfied. A trail \( M \) is a list of literals that are set to true, which can be decided literals (or decisions) and implied literals (or implications). Decisions are guesses made by the DPLL algorithm by applying the split rule. A decided literal \( l \) on the trail is denoted by \( \bullet l \). Implications are produced by applications of the unit propagation rule. We can think of implications as informed decisions, since we know the reason why the implied literal needs to be true. The \text{value} \ function is used to evaluate a literal under the trail:

\[
\text{value}(l, M) = \begin{cases} 
\text{true} & l \in M \\
\text{false} & -l \in M \\
\text{undef} & \text{otherwise}
\end{cases}
\]

We can overload it to evaluate a clause \( c \) under a trail \( M \):

\[
\text{value}(c, M) = \begin{cases} 
\text{true} & \text{there exists } l \in c \text{ s.t. } \text{value}(l, M) = \text{true} \\
\text{false} & \text{for every literal } l \in c, \text{value}(l, M) = \text{false} \\
\text{undef} & \text{otherwise}
\end{cases}
\]

We say that a trail \( M \) is complete if every literal \( l \) in the formula \( C \) has a value under the trail.

Checking the satisfiability of formula (i.e. multiset of clauses) \( C_0 \) in CNF starts in the state \( \langle [], C_0 \rangle \) and the search is expected to reach one of the accepting states: 1. \text{unsat} if the \( C_0 \) is unsatisfiable, or 2. \text{sat} if the \( C_0 \) is satisfiable

The backtracking rule corresponds to the return from a recursive call when applying the split rule in the algorithmic description. The rule backtracks only the last decision on the trail.

5.1.3 The CDCL algorithm

In subsequent years, the implementation and understanding of the DPLL algorithm has reached new levels. Between some crucial insights and highly
specialized data structures, the algorithm has become even more streamlined and efficient. We switch now completely to the abstract transition system notation and observe how we can improve the rules of the system.

The conflict-driven clause learning (CDCL) algorithm has two crucial improvements: conflict analysis and learning.

The DPLL algorithm represents an exhaustive search with the pruning of the search space done only through propagation. However, one can do better than just backtracking one decision at the time. A clause falsified by the current partial assignment is called a conflict clause and its discovery a conflict. Each literal on the trail is either decided or implied. When a conflict occurs, instead of backtracking immediately, the algorithm attempts to repair the partial assignment in a meaningful way by jumping back to a decision relevant to the conflict. This is done by keeping reasons for the implied literals on the trail, in the form of clauses that implied them. Once a conflict is discovered, we can apply boolean resolution \cite{29} to the conflict clause and the implication clause to discover a new conflict clause. The backtracking is performed by resolving the conflict clause against explanations of trail elements. At some point during the backtracking, the conflict clause either becomes an implication or there are no more decisions on the trail.

---

**Propagate**

\[
\langle M, C \rangle \rightarrow \langle [M, l], C \rangle \quad \text{if} \quad \forall i : \text{value}(l_i, M) = \text{false} \\
\quad \text{value}(l, M) = \text{undef}
\]

**Decide**

\[
\langle M, C \rangle \rightarrow \langle [M, \bullet l], C \rangle \quad \text{if} \quad \text{value}(l, M) = \text{undef}
\]

**Sat**

\[
\langle M, C \rangle \rightarrow \text{sat} \quad \text{if} \quad M \text{ is complete} \\
\quad \text{value}(c, M) = \text{true} \text{ for all } c \in C
\]

**Unsat**

\[
\langle M, C \rangle \rightarrow \text{unsat} \quad \text{if} \quad M \text{ contains no decisions} \\
\quad \text{value}(c, M) = \text{false} \text{ for some } c \in C
\]

**Backtrack**

\[
\langle [M, \bullet l, N], C \rangle \rightarrow \langle M, \lnot l, C \rangle \quad \text{if} \quad N \text{ contains no decisions} \\
\quad \text{value}(c, M) = \text{false} \text{ for some } c \in C
\]
Implementing this advanced form of backtracking is implemented using the rules: *consume*, *resolve* and *backjump*.

The clauses synthesized through the conflict analysis are always implied by the problem. Conflicts involving the same combination of literals can be encountered at multiple points in the search space. To preempt their repeated resolution, we can add the derived clause to the formula. Adding such a clause is called *learning*. Over time learned clause can become detrimental due to upkeep overhead, so we can choose to drop some of them, i.e., *forget* them. The *learn* and *forget* rules describe conditions that need to be met. In order to ensure termination of the overall algorithm, applications of these rules are usually done with caution.

The last addition over the DPLL algorithm are the restarts. If the goal is to find a satisfying assignment for the formula, restarting the search with additional information might be beneficial. The added information comes in the form of learned clauses. The *restart* rule allows the search to continue from an empty partial model. This rule is dangerous in the sense that it can affect termination and completeness of the algorithm. It also opens up the space for yet another set of heuristics, in terms of strategy of applying the restart rule. The Luby restart scheme is one of the most successful and well-known, but others schemes exist [20].

The success of the CDCL algorithm is also in part due to highly efficient data structures that allow easy backtracking of the trail with little to no upkeep, as well as efficient detection of propagations in the form of the *two-watch literal scheme*. Also, a number of heuristics plays a crucial role in the performance of the algorithm [4], such as the choice of variable to decide upon (e.g., VSDIS heuristic), which clauses to learn and which to forget, how often should the search be restarted.

### The CDCL transition system

The CDCL transition system is similar to that of the DPLL transition system, with some modifications. We introduce a new type of state to represent conflicts, so the system has the following states:

- accepting states *unsat* and *sat*
- search states $\langle M, C \rangle$
- conflict state $\langle M, C \rangle \vdash c$

In order to support conflict analysis, the trail is modified to consist of two kinds of elements:

- *decided literals*, denoted by $\bullet l$
• implied literals, denoted by \( e \rightarrow l \), where \( e \) is the implication clause and \( l \) the implied literal.

The rules are shown in Fig. 5.2. The main difference to the DPLL transition system is that the rules can be divided into two groups: clausal search rules and conflict analysis rules. Clausal search rules are very much like those found in the DPLL transition system. The conflict analysis rules (\( \text{conflict, consume, resolve, backjump} \)) replace the backtrack rule and provide the algorithm with a powerful correcting mechanism when the search goes astray.

5.2 The SMT Problem

While many problems can be encoded directly in propositional logic, there exist more natural ways to encode them in different theories. The satisfiability modulo theories (SMT) problem is the problem of determining satisfiability of the formula in the presence of background theories. The background theories could be various, depending on the nature of the problem, e.g., equality with uninterpreted functions, difference logic, theory of linear arithmetic, arrays, bit-vectors, floating-point arithmetic, strings and so on [14]. The advantage of modeling a problem using SMT rather than SAT is usually in of expressiveness, ease of modeling and scalability (many encodings into propositional logic have a non-linear spatial complexity).

There are a number of approaches to the SMT problem. The straightforward way is called eager SMT and entails translation of the problem into an equisatisfiable propositional formula, and using a SAT solver [22]. This is a relatively simple approach, but it has the downside of encoding everything upfront, which can be a problem for a number of theories, such as BVA and FPA where the resulting formula can easily be too large for the SAT solver to even start solving. UCLID [23] is an example of an eager SMT solver.

Another approach is lazy SMT, where the formula is abstracted into Boolean and theory worlds. A SAT solver is used to enumerate satisfying assignments to the Boolean structure, and a theory-specific procedure is used to determine whether that assignment is consistent within the theory. The theory procedure guides the search of the SAT solver. In addition, theory consistency can be checked against the partial models in order to discover inconsistency as soon as possible and theory knowledge can be learned in the propositional world (avoiding repeated theory reasoning). It can also leverage restarts and backtracking successfully.

The advantages of the lazy approach is that by keeping the boolean and theory world separated, the solver can be engineered in a modular way. The solvers communicate through a simple interface. The theory solver must
be able to determine satisfiability of a conjunction of theory literals. The advantage of a dedicated theory solver is that one can use tailor-made procedures suitable for the problem at hand, rather than relying on the ability of the SAT solvers to handle the encoding. Especially since encodings themselves can lose structural information, e.g., encoding BVA into propositional logic loses the word level information. It is possible to have multiple theory solvers which cooperate in the case of theory combinations. This approach is also known as the DPLL(T), which reflects the fact that the outer search procedure is a SAT solver, that relies on a theory solver to guide the search. It also reflects modularity in an elegant way, since a different theory solver yields a new SMT procedure altogether.

**Transition system of DPLL(T)**

The DPLL(T) transition system adds *theory rules* to the CDCL transition system, shown in Fig. 5.3. The *T-conflict* rule detects when a set of literals on the trail is inconsistent in the theory $T$ and transitions to a conflict state. The *T-propagate* rule detects when a literal is implied in the theory $T$ and asserts that literal in the Boolean world. Presented like this, the conflict analysis is offloaded to the Boolean conflict analysis rules entirely. A number of different procedures is obtained by varying the strategies of rule application. For more details are available in the original paper on abstract DPLL(T) [28].

**5.2.1 Beyond DPLL(T)**

In recent years two approaches similar and yet more general than DPLL(T) have emerged.

The model constructing satisfiability calculus (mcSAT)[15] is a more general framework that encompasses both DPLL(T) and the model constructing approaches. It is a flexible framework that can be instantiated in either direction depending on the strategies taken during the search. The main difference to the DPLL(T) framework is that it allows direct model assignments on the trail. It takes a step further in detecting infeasibility as early as possible, through the requirement that the trail should be a partial model at every point in time and that all the theory literals on the trail should be satisfied by model assignments before new Boolean decisions are made. Another advantage that it offers, and that DPLL(T) implementations do not really attempt to leverage, is the ability to return theory lemmas as explanations of a conflict which feature newly introduced literals. This allows true theory learning to take place.

Abstract conflict driven clause learning (ACDCL) [17] is the other ap-
The main idea is to combine the CDCL algorithm with a lattice abstraction of theory predicates. The lattice can then be used dually throughout the search, as a form of splitting-on-demand (by moving downwards through the lattice), but also as a form of generalization during explanation generalization (by following it upwards).
5.2. The SMT Problem

Propagate

\[ \langle M, C \rangle \quad \rightarrow \quad \langle [M, c \rightarrow l], C \rangle \quad \text{if} \quad \forall i : value(l_i, M) = false \]

\[ value(l, M) = undef \]

Decide

\[ \langle M, C \rangle \quad \rightarrow \quad \langle [M, \bullet l], C \rangle \quad \text{if} \quad l \in B, value(l, M) = undef \]

Conflict

\[ \langle M, C \rangle \quad \rightarrow \quad \langle M, C \rangle \models c \quad \text{if} \quad c \in C, value(c, M) = false \]

Sat

\[ \langle M, C \rangle \quad \rightarrow \quad sat \quad \text{if} \quad M \text{ is complete} \]

\[ \forall c \in C : value(c, M) = true \]

Unsat

\[ \langle M, C \rangle \quad \rightarrow \quad unsat \quad \text{if} \quad M \text{ contains no decisions} \]

\[ \exists c \in C : value(c, M) = false \]

Resolve

\[ \langle [M, d \rightarrow l], C \rangle \models c \quad \rightarrow \quad \langle M, C \rangle \models r \quad \text{if} \quad \neg l \in c \]

\[ r = resolve(c, d, l) \]

Consume

\[ \langle [M, d \rightarrow l], C \rangle \models c \quad \rightarrow \quad \langle M, C \rangle \models c \quad \text{if} \quad \neg l \notin c \]

\[ \langle [M, l], C \rangle \models c \quad \rightarrow \quad \langle M, C \rangle \models c \quad \text{if} \quad \neg l \notin c \]

Backjump

\[ \langle [M, N], C \rangle \models c \quad \rightarrow \quad \langle [M, c \rightarrow l], C \rangle \quad \text{if} \quad c = l_1 \lor \ldots \lor l_m \lor l \]

\[ \forall i : value(L_i, M) = false \]

\[ value(l, M) = undef \]

\[ N \text{ starts with a decision} \]

Learn

\[ \langle M, C \rangle \models c \quad \rightarrow \quad \langle M, C \cup \{c\} \rangle \models c \quad \text{if} \quad c \notin C \]

Forget

\[ \langle M, C \rangle \quad \rightarrow \quad \langle M, C \setminus \{c\} \rangle \quad \text{if} \quad c \in C \text{ is a learned clause} \]

Restart

\[ \langle M, C \rangle \quad \rightarrow \quad \langle [], C \rangle \quad \text{if} \]

Figure 5.2: The rules of the CDCL transition system
T-Propagate

\[ \langle M, C \rangle \rightarrow \langle [M, e \rightarrow l], C \rangle \text{ if } \]

\[
\begin{align*}
& l_1, l_2, \ldots, l_n \in M \\
& l_1, l_2, \ldots, l_n \models_T l \\
& e = (-l_1 \lor -l_2 \lor \ldots \lor -l_n \lor l)
\end{align*}
\]

T-Conflict

\[ \langle M, C \rangle \rightarrow \langle M, C \rangle \vdash c \text{ if } \]

\[
\begin{align*}
& l_1, l_2, \ldots, l_n \in M \\
& l_1, l_2, \ldots, l_n \models_T \bot \\
& c = (-l_1 \lor -l_2 \lor \ldots \lor -l_n)
\end{align*}
\]

Figure 5.3: Theory rules of DPLL(T)
Chapter 6

Reasoning about Machine Arithmetic

In this chapter an overview of existing decision procedures for reasoning about machine arithmetic. They focus on two theories, the fixed-size bit-vectors and the floating-point arithmetic.

6.1 Decision procedures for bit-vectors

The theory of bit-vector arithmetic is used to describe the behaviour and operations of bounded integer arithmetic. The data type itself is considered to have string-like nature, where values are obtained by appending bit constants, zeroes and ones, together to form values. The data type is parametric in size and has three types of operations: 1. string operations (concatenation and extraction) 2. logical operations (bitwise operations), and 3. arithmetic operations (addition, subtraction, etc) Their semantics matches the (now) standard implementation of bounded integer arithmetic, with a few notable exceptions where a consensus is yet to be reached by the SMT community.

6.1.1 Bit-Blasting

The straight-forward approach is to encode the BVA constraints into the propositional logic following the semantics of the operations (this is identical to the correct implementation of the corresponding hardware circuits). This procedure is called flattening or bit-blasting [22]. For each bit-vector variable $v$ of size $l$, $l$ new Boolean variables are introduced $v_0, v_1, \ldots, v_{l-1}$. Atomic constraints of bit-vector arithmetic are now expressed in terms of their equivalent bit variables (following their hardware circuit implementation for example).
This approach is very simple, it simply mimics what the hardware does. However, it scales rather poorly, since the constraints might involve a number of auxiliary propositional variables and constraints that is quadratic in size of the bit-vector. The resulting formula can be so big that SAT solver cannot fit it into memory. Another downside is that by encoding it into propositional logic, word level information is lost since the SAT solver knows nothing about grouping of variables into bit-vectors, for example.

6.1.2 Pre-Processing and In-Processing

In order to make the formula easier for the SAT solver, the formula is simplified using transformations that are known to be correct. This kind of approach is called pre-processing. Additionally, we might apply similar rules during on the fly, during the solving of the formula. In that case it is called in-processing.

6.1.3 Lazy and Eager Approaches

Recently an approach that combines different solvers together emerged [18]. It’s a DPLL(T)-style solver that combines multiple solvers specialized for different fragments of the BVA theory, such as the core theory of extraction and concatenation or the theory of bit-vector inequalities. These solvers complement one another, so a portfolio approach yields an overall efficient solver.

6.2 Decision procedures for floating-point arithmetic

Reasoning about the floating-point arithmetic is tricky, because of so many layers involved. The code, the compiler and the hardware could each affect behavior of a floating-point program. The SMT-LIB theory of floating-point arithmetic was written to reflect the semantics defined by the IEEE-754 standard. There are two decision procedures of note for the FPA theory. One is a combination of encodings into BVA and the propositional logic. The other one uses abstract conflict driven clause learning.

6.2.1 Encoding into bit-vector arithmetic

A straightforward way to approach solving constraints of the FPA theory, is to encode floating-point variables $v$ as a triplet of bit-vector variables $(v_{\text{sgn}}, v_{\text{exp}}, v_{\text{sig}})$ and translate floating-point operations into corresponding
bit-vector constraints over the triples of bit-vectors following their the IEEE-754 standard compliant hardware implementation. The resulting bit-vector formula is then bit-blasted and given to a SAT solver.

6.2.2 The ACDCL approach

The ACDCL algorithm [6] uses an interval abstraction for each floating-point variable and leverages the properties of the interval domain that are compatible with the primitives of the CDCL algorithm. The intervals can be split (in half for example), which matches the splitting-on-demand features of the CDCL algorithm. Additionally intervals can quite easily be shown to be disjoint, making them particularly suitable to proving unsatisfiability. The downside is that even though a formula is proven satisfiable, it might take some time before the model is available. This can be problematic for applications that are dependent on retrieving the model, such as test case generation.
Chapter 7

Contributions

The work presented in this thesis improves upon existing procedures for reasoning about machine arithmetic.

In recent years, approximations and abstractions have been successfully applied to reason about various domains. To ensure soundness and completeness, these algorithms require that approximations have certain conservative properties. Approximations that have those properties are called over- and under-approximations. However, for certain domains, such as floating-point arithmetic, coming up with approximations that have these properties is particularly difficult. As a consequence, approaches relying on over- and under-approximations have proven to be ineffective for floating-point arithmetic. Approximations which are not over- nor under-approximations can find incorrect solutions or overlook correct solutions. Paper I presents a general approximation framework which does not require approximations to have any particular properties. An existing procedure is used to reason about the approximations, and if it is sound, complete and terminating, the framework preserves these properties. Abstraction of floating-point numbers using reduced-precision floating-point numbers has been implemented. The framework in combination with this abstraction has improved the performance of the bit-blasting-based procedure, yielding speed-ups of an order of magnitude in some cases and solving a larger number of formulas overall.

An alternative to the DPLL(T)-style SMT solvers are model constructing procedures. The model constructing satisfiability calculus (mcSAT)[15] was proposed as a general framework for defining new model constructing procedures. The advantage of the model constructing approach is the native domain reasoning. An obvious example is the bit-vector theory which is commonly encoded into the less expressive propositional logic. While this gains the use of highly efficient SAT solvers, the encoding loses structural information needed to reason about some aspects efficiently (e.g., arithmetic
operations). Paper II presents a novel model constructing decision procedure for the theory of bit-vectors, called mcBV. It uses a lazy representation of bit-vectors, attempting to avoid bit-blasting altogether. It features two different abstractions of bit-vectors, each targeting a particular class of operations. The bit-pattern abstractions are suited to string-like and logic operations, while the interval abstraction is used to reason about arithmetic properties. These abstractions allow mcBV to reason about unusually large bit-vectors. A greedy explanation generalization algorithm is also presented, which offers a new powerful way of learning explanations. Our prototype implementation of mcBV offers promising experimental results on several classes of formulas, offering order of magnitude speedups on some instances. mcBV offers a complementary approach to bit-blasting, especially when it comes to reasoning about large bit-vectors.
Chapter 8

Conclusion

The work presented in this thesis offers two novel approaches to reasoning in domains relevant for machine arithmetic. In both cases, coming up with the right abstractions for the particular domain has proven crucial for the procedure. Finding those abstraction is far from easy. However, if their properties are exploited, results are significant improvements in performance. To make experimentation with abstractions easier, the approximation refinement framework can be implemented as an abstract solver. This would enable easy swapping of the underlying decision procedure, which would allow the user to check how a particular abstraction interacts with a particular solving technology. Additionally, if provided with a language for describing approximations, the framework could become a test bed for prototyping new decision procedures. A different direction of future work is a model-constructing procedure for the theory of floating-point arithmetic. Given the lack of native domain reasoning procedures for floating-point arithmetic, such a procedure might yield a complementary approach to the existing methods.
Bibliography


Paper I
Approximations for Model Construction

Aleksandar Zeljić · Christoph M. Wintersteiger · Philipp Rümmer

Abstract We consider the problem of automatically and efficiently computing models for satisfiable constraints, in the presence of complex background theories such as floating-point arithmetic. Model construction has various applications, for instance for automatic generation of test inputs. It is well-known that a naïve encoding of constraints into simpler theories (for instance, bit-vectors or propositional logic) often leads to a drastic increase in size, or that it is unsatisfactory in other terms of space and runtime required for model construction. We define a framework for systematic application of approximations in order to improve model construction performance. Our method is more general than previous techniques in the sense that approximations that are neither under- nor over-approximations can be used, and it shows promising performance on practically relevant benchmark problems.

1 Introduction

The construction of satisfying assignments (or, more generally, models) for a set of given constraints is one of the most central problems in automated reasoning. Although the problem has been addressed extensively in research fields including constraint programming and more recently in satisfiability modulo theories (SMT), there are still constraint languages and background theories where effective model construction is challenging. Such theories are, in particular, arithmetic domains such as bit-vectors, nonlinear real arithmetic (or real-closed fields), and floating-point arithmetic (FPA); even when decidable, the high computational complexity
of such languages turns model construction into a bottleneck in applications such as bounded model checking, white-box test case generation, analysis of hybrid systems, and mathematical reasoning in general.

We follow a recent line of research that applies the concept of abstraction to model construction (e.g., [3,5,10,19]). In this setting, constraints are usually simplified prior to solving to obtain over- or under-approximations, or some combination thereof (mixed abstractions); experiments have shown that this concept can speed up model construction significantly. However, previous work in this area suffers from the fact that the definition of good over- and under-approximations is difficult and limiting, for instance in the context of floating-point arithmetic. We argue that the focus on over- and under-approximations is neither necessary nor optimal: as a more flexible alternative, we present a general algorithm that is able to incorporate any form of approximation in the model construction process, including approximations that cannot naturally be represented as a combination of over- and under-approximations. Our method preserves essential properties like soundness, completeness, and termination.

For the purpose of empirical evaluation, we instantiate our model construction procedure for the domain of floating-point arithmetic, and present an evaluation based on an implementation thereof within the Z3 theorem prover [22]. Experiments on practically relevant and satisfiable floating-point benchmark problems (SMT-LIB QF_FP) show an average speed-up of roughly one order of magnitude when compared to the naive bit-blasting-based default decision procedure that comes with Z3. Further experiments show that the performance of our prototype implementation is also competitive with other state-of-the-art solvers for floating-point arithmetic.

While mainly intended for model generation, our prototype also solves unsatisfiable problems, and thanks to a new technique for refinement of unsatisfiable (sub-)problems, only a small performance penalty is incurred on them. However, we believe that further research is necessary to improve reasoning for unsatisfiable problems, even though our current prototype implementation exhibits satisfactory performance on unsatisfiable benchmark problems.

The contributions of this article are as follows:

1. a general method for model construction based on (non-conservative) approximations,
2. an instantiation of our method for the theory of floating-point arithmetic,
3. refinement techniques for approximate models and unsatisfiable problems, as well as
4. an experimental evaluation of a prototype implementation of all proposed methods.

1.1 Motivating Example

To illustrate our motivation and the resulting techniques, consider a heavily simplified software proportional-integral (PI) controller operating on floating-point data, as shown in Alg. 1. All variables in this example range over double precision (64-bit) IEEE-754 floating-point numbers. The controller is initialized with the set_point value and
## 1.1 Motivating Example

Algorithm 1: Software PI controller

```c
1 const double Kp=1.0;
2 const double Ki=0.25;
3 const double set_point=20.0;
4 double integral = 0.0;
5 double error;
6
7 for (int i = 0; i < N; ++i) {
8     in = read_input();
9     error = set_point - in;
10    integral = integral + error;
11    out = Kp*error + Ki*integral;
12    set_output(out);
13 }
```

the constants $K_p$ and $K_i$, it reads input values (in; e.g., from a sensor) via function `read_input`, and it computes output values (out) which control the system through the function `set_output`. The controller computes the control values in such a way, that the input values are as close to `set_point` as possible. For simplicity, we assume that there is a bounded number $N$ of control iterations.

Suppose we want to prove that if the input values stay within the range $18.0 \leq \text{in} \leq 22.0$, then the control values will stay within a range that we consider safe, for instance $-3.0 \leq \text{out} \leq +3.0$. This property is true of our controller only for two control iterations, but it can be violated within three.

A bounded model checking approach to this problem produces a series of formulas, one for each $N$ and it then checks the satisfiability of those formulas (usually in sequence). Today, most (precise) solvers for floating-point formulas implement this satisfiability check by means of bit-blasting, i.e., using a bit-precise encoding of FPA semantics as a propositional formula. Due to the complexity of FPA, the resulting formulas grow very quickly, and tend to overwhelm even the fastest SAT/SMT solvers. For example, an unrolling of the PI controller example to $N=100$ steps cannot be solved by Z3 within an hour of runtime (see Tbl. 1).

<table>
<thead>
<tr>
<th>Bound $N$</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clauses ($\times 10^4$)</td>
<td>96</td>
<td>230</td>
<td>630</td>
<td>1298</td>
<td>2633</td>
<td>3969</td>
<td>5304</td>
<td>6639</td>
<td>13316</td>
</tr>
<tr>
<td>Variables ($\times 10^3$)</td>
<td>12</td>
<td>28</td>
<td>78</td>
<td>161</td>
<td>326</td>
<td>492</td>
<td>657</td>
<td>822</td>
<td>1649</td>
</tr>
<tr>
<td>Z3 time (s)</td>
<td>1</td>
<td>5</td>
<td>19</td>
<td>27</td>
<td>288</td>
<td>1190</td>
<td>1962</td>
<td>3297</td>
<td>$&gt;1h$</td>
</tr>
</tbody>
</table>

Table 1: Behavior of Z3 on the PI controller example.

However, this example has the property that the full range of floating-point numbers is not required to find suitable program inputs; essentially a prover just needs to find a sequence of inputs such that the errors add up to a sum that is greater than 3.0. There is no need to consider numbers with large magnitude, or a large number of significand digits/bits. We postulate that this situation is typical for many practical applications. Since bit-precise treatment of floating-
point numbers is clearly wasteful in this setting, we might consider some of the following alternatives:

- all operations in the program can be evaluated in \textbf{real} instead of floating-point arithmetic. For problems with only linear operations, such as the program at hand, this enables the use of highly efficient solvers based on linear programming (LP). However, the straight-forward encoding into LP would ignore the possibility of overflows or rounding errors. A bounded model checking approach based thereupon will therefore be neither sound nor complete. Further, little is gained in terms of computational complexity for nonlinear constraints.

- operations can be evaluated in \textbf{fixed-point} arithmetic. Again, this encoding does not preserve the overflow- and rounding-semantics of FPA, but it enables solving using more efficient bit-vector encodings and solvers.

- operations can be evaluated in FPA with \textbf{reduced precision}: we can use single precision numbers, or other formats even smaller than that.

Strictly speaking, soundness and completeness are lost in all three cases, since the precise nature of overflows and rounding in FPA is ignored. All three methods enable, however, the efficient computation of \textit{approximate models}, which are likely to be “close” to genuine double-precision FPA models, for some notion of closeness. In this paper, we define a general framework for model construction with approximations. In order to establish soundness and completeness of our model construction algorithm, the framework contains a \textbf{model reconstruction} phase, in which approximate models are translated into precise models. This reconstruction may fail, in which case \textit{approximation refinement} is used to iteratively increase the precision of approximate models.

\section*{2 Related Work}

Related work to our contribution falls into two categories: general abstraction and approximation frameworks, and specific decision procedures for floating-point arithmetic.

The concept of abstraction (and approximation) is central to software engineering and program verification, and it is increasingly employed in general mathematical reasoning and in decision procedures. Usually, and in contrast to our work, only under- and over-approximations are considered, i.e., the formula that is solved either implies or is implied by an approximate formula (or abstraction). Counter-example guided abstraction refinement \cite{counterexample guided abstraction refinement} is a general concept that is applied in many verification tools and decision procedures (e.g., even on a relatively low level like in QBF \cite{QBF} or in model based quantifier instantiation for SMT \cite{model based quantifier instantiation}).

A general framework for abstracting decision procedures is Abstract CDCL, recently introduced by D'Silva et al. \cite{abstract CDCL}, which was also instantiated with great success for FPA \cite{FPA decision procedure}. This approach relies on the definition of suitable abstract domains for constraint propagation and learning. In our experimental evaluation, we compare to the FPA decision procedure in MathSAT, which is an instance of ACDCL. ACDCL can also be integrated with our framework, e.g., to solve approximations. A further framework for abstraction in theorem proving was proposed by Giunchiglia et al. \cite{Giunchiglia et al.}. Again, this work focuses on under- and over-approximations, not on other forms of approximation.
Specific instantiations of abstraction schemes in related areas include the bit-vector abstractions by Bryant et al. [5] and Brummayer and Biere [4], as well as the (mixed) floating-point abstractions by Brillout et al. [3]. Van Khanh and Ogawa present over- and under-approximations for solving polynomials over reals [19]. Gao et al. [12] present a δ-complete decision procedure for nonlinear reals, considering over-approximations of constraints by means of δ-weakening.

There is a long history of formalization and analysis of FPA concerns using proof assistants, among others in Coq by Melquiond [21] and in HOL Light by Harrison [15]. Coq has also been integrated with a dedicated floating-point prover called Gappa by Boldo et al. [1], which is based on interval reasoning and forward error propagation to determine bounds on arithmetic expressions in programs [9]. The ASTREE static analyzer [8] features abstract interpretation-based analyses for FPA overflow and division-by-zero problems in ANSI-C programs. The SMT solvers MathSAT [6], Z3 [22], and Sonolar [20], all feature (bit-precise) conversions from FPA to bit-vector constraints.

3 Preliminaries

We establish a formal basis in the context of multi-sorted first-order logic (e.g., [16]). A signature \( \Sigma = (S, P, F, \alpha) \) consists of a set of sort symbols \( S \), a set of sorted predicate symbols \( P \), a set of sorted function symbols \( F \), and a sort mapping \( \alpha \). Each predicate and function symbol \( g \in P \cup F \) is assigned a \((k+1)\)-tuple \( \alpha(g) \) of argument sorts (with \( k \geq 0 \)), where \( k \) is the arity of the symbol. Constants are considered to be nullary function symbols. Also, the Boolean sort symbol is included in the set of sorts, i.e. \( s_b \in S \). We assume a countably infinite set \( X \) of variables, and (by abuse of notation) overload \( \alpha \) to assign sorts also to variables. Given a multi-sorted signature \( \Sigma \) and variables \( X \), the notions of well-sorted terms, atoms, literals, clauses, and formulas are defined as usual. The function \( \text{fv}(\phi) \) denotes the set of free variables in a formula \( \phi \). In what follows, we assume that all formulas are quantifier-free.

A \( \Sigma \)-structure \( m = (U, I) \) with underlying universe \( U \) and interpretation function \( I \) maps each sort \( s \in S \) to a non-empty set \( I(s) \subseteq U \), each predicate \( p \in P \) of sorts \( (s_1, s_2, \ldots, s_k) \) to a relation \( I(p) \subseteq I(s_1) \times I(s_2) \times \ldots \times I(s_k) \), and each function \( f \in F \) of sort \( (s_1, s_2, \ldots, s_k, s_{k+1}) \) to a set-theoretic function \( I(f) : I(s_1) \times I(s_2) \times \ldots \times I(s_k) \to I(s_{k+1}) \). A variable assignment \( \beta \) under a \( \Sigma \)-structure \( m \) maps each variable \( x \in X \) to an element \( \beta(x) \in I(\alpha(x)) \). The valuation function \( \text{val}_{m,\beta}(\cdot) \) is defined for terms and formulas in the usual way. A theory \( T \) is a pair \( (\Sigma, M) \) of a multi-sorted signature \( \Sigma \) and a class of \( \Sigma \)-structures \( M \). A formula \( \phi \) is \( T \)-satisfiable if there is a structure \( m \in M \) and a variable assignment \( \beta \) such that \( \phi \) evaluates to \text{true} \}; we denote this by \( m, \beta \models_T \phi \), and call \( \beta \) a \( T \)-solution of \( \phi \).

4 The Approximation Framework

We describe a model construction procedure for formulas \( \phi \) over a set of variables \( X \) in a theory \( T \). The goal is to obtain a \( T \)-solution of \( \phi \). The main idea underlying our method is to replace the theory \( T \) with an approximation theory \( \hat{T} \), which
enables explicit control over the precision used to evaluate theory operations. In our method, the $T$-problem $\phi$ is first lifted to a $\hat{T}$-problem $\hat{\phi}$, then solved in the theory $\hat{T}$, and finally, if a solution is found, it is translated back to a $T$-solution. The benefit of using the theory $\hat{T}$ is that different levels of approximation may be used during computation. We will use the theory of floating-point arithmetic as a running example for instantiation of this framework.

\[ \hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi} \]

\[ m, \beta \models_T \phi \]

Fig. 1: Commutativity graph showing how the model $m, \beta$ can be obtained via approximation theory $\hat{T}$.

4.1 Approximation Theories

In order to formalize the approach of finding models by means of approximation, we construct the approximation theory $\hat{T} = (\hat{\Sigma}, \hat{M})$ from $T$, by extending all function and predicate symbols with a new argument representing the precision to which the function or predicate should be computed.

**Syntax.** We introduce a new sort for the precision $s_p$, and a new predicate symbol $\preceq$ which orders precision values. The signature $\hat{\Sigma} = (\hat{\Sigma}, \hat{P}, \hat{F}, \hat{\alpha})$ is obtained from $\Sigma$ in the following manner: $\hat{\Sigma} = \Sigma \cup \{s_p\}$; the set of predicate symbols is extended with the new predicate symbol $\preceq$, $\hat{P} = P \cup \{\preceq\}$; the set of function symbols is extended with the new constant $\omega$, representing the maximum precision value, $\hat{F} = F \cup \{\omega\}$; the sort function $\hat{\alpha}$ is defined as

\[ \hat{\alpha}(g) = \begin{cases} (s_p, s_1, s_2, \ldots, s_n) & \text{if } g \in P \cup F \text{ and } \alpha(g) = (s_1, s_2, \ldots, s_n) \\ (s_p, s_p, s_b) & \text{if } g = \preceq \\ (s_p, s_p) & \text{if } g = \omega \\ \alpha(g) & \text{otherwise} \end{cases} \]

Note that constant symbols become unary function symbols instead.

**Semantics.** $\hat{\Sigma}$-structures $(\hat{U}, \hat{I})$ enrich the original $\Sigma$-structures by providing approximate versions of function and predicate symbols. The resulting operations may be under- or over-approximations, but they may also be approximations that are close to the original operations’ semantics by some other metric. The degree of approximation is controlled with the help of the precision argument. We assume that the set $\hat{M}$ of $\hat{\Sigma}$-structures satisfies the following properties:
for every structure \( (\hat{U}, \hat{I}) \in \hat{M} \), the relation \( \hat{I}(\preceq) \) is a partial order on \( \hat{I}(s_p) \)
that satisfies the ascending chain condition (every ascending chain is finite),
and that has the unique greatest element \( \hat{I}(\omega) \in \hat{I}(s_p) \);
for every structure \( (U, I) \in M \), an approximation structure \( (\hat{U}, \hat{I}) \in \hat{M} \) extending \( (U, I) \) exists,
together with an embedding \( h : U \mapsto \hat{U} \) such that, for every sort \( s \in S \), function \( f \in F \), and predicate \( p \in P \):
\[
h(I(s)) \subseteq \hat{I}(s)
\]
\[
(a_1, \ldots, a_n) \in I(p) \iff (\hat{I}(\omega), h(a_1), \ldots, h(a_n)) \in \hat{I}(p) \quad (a_i \in I(\alpha(p)_i))
\]
\[
h(I(f)(a_1, \ldots, a_n)) = \hat{I}(f)(\hat{I}(\omega), h(a_1), \ldots, h(a_n)) \quad (a_i \in I(\alpha(f)_i))
\]
– vice versa; for every approximation structure \( (\hat{U}, \hat{I}) \in \hat{M} \) there is a structure \( (U, I) \in M \) that is similarly embedded in \( (\hat{U}, \hat{I}) \).

These properties ensure that every T-model has a corresponding \( \hat{T} \)-model, i.e.
that no models are lost. Interpretations of function and predicate symbols under \( \hat{I} \)
with maximal precision are isomorphic to their original interpretation under \( I \). The
interpretation \( \hat{I} \) should interpret the function and predicate symbols in such a way
that their interpretations for a given value of the precision argument approximate
the interpretations of the corresponding function and predicate symbols under \( I \).
And finally, that it is possible to translate every \( \hat{T} \)-model into some T-model, using
a mapping \( h^{-1} \) that reverses the embedding \( h \) (not necessarily its mathematical
inverse, since \( h \) is rarely going to be bijective, but an inverse in spirit).

4.2 Application to Floating-Point Arithmetic

The IEEE-754 standard for floating-point numbers [17] defines floating-point numbers,
their representation in bit-vectors, and the corresponding operations. Most
crucially, bit-vectors of various sizes are used to represent the significand and the
exponent of numbers; e.g., double-precision floating-point numbers are represented
by using 11 bits for the exponent and 53 bits for the significand. denote the subset
of reals that can be represented as floating-point numbers \( s \) significand bits and
e exponent bits by \( FP_{s,e} \):
\[
FP_{s,e} = \left\{ (-1)^{sgn} \cdot \text{sig} \cdot 2^{exp-s} \mid \begin{array}{l}
sgn \in \{0, 1\}, \\
\text{sig} \in \{0, \ldots, 2^s - 1\}, \\
exp \in \{-2^{e-1} + 3, \ldots, 2^{e-1}\}
\end{array} \right\} \cup \{\text{NaN}, +\infty, -\infty, -0\}
\]

The set consists of: 1. normalized numbers (in practice encoded with an implicit
leading bit set to 1), 2. subnormal numbers, and 3. special values. The definition
does not discriminate between normal and subnormal numbers and any value with
multiple representations loses the multiplicity in the set. Since the reals do not contain
a signed zero value it is included explicitly with the other special values.

**Proposition 1 (Inclusion property)** FP domains grow monotonically when
increasing \( e \) or \( s \), i.e., \( FP_{s',e'} \subseteq FP_{s,e} \) provided that \( s' \leq s \) and \( e' \leq e \); we call
this the inclusion property.
For fixed values $e$ of exponent bits and $s$ of significand bits, FPA can be modeled as a theory in our sense. We denote this theory by $TF_{s,e}$, and write $s_f$ for the sort of FP numbers, and $s_r$ for the sort of rounding modes. The various FP operations are represented as functions and predicates of the theory; for instance, floating-point addition turns into the function symbol $\oplus$ with signature $\alpha(\oplus) = (s_r, s_f, s_f, s_f)$. Additional constants of sort $s_r$ are provided for the five rounding modes in the IEEE-754 standard, namely

- RoundTowardZero,
- RoundNearestTiesToEven,
- RoundNearestTiesToAway,
- RoundTowardPositive, and
- RoundTowardNegative.

The semantics of $TF_{s,e}$ is defined by a single structure $(U_{s,e}, I_{s,e})$ with $I_{s,e}(s_f) = FP_{s,e}$. The semantics of floating-point operations is derived from the corresponding operations over reals, except in cases where the resulting values are not representable as floating-point numbers; then rounding takes place in accordance with the chosen rounding mode.

**FPA approximation theories.** We construct the approximation theory $\hat{TF}_{s,e}$, by introducing the precision sort $s_p$, predicate symbol $\preceq$, and a constant symbol $\omega$. The function and predicate symbols have their signature changed to include the precision argument. For example, the signature of the floating-point addition symbol $\oplus$ is $\hat{\alpha}(\oplus) = (s_p, s_r, s_f, s_f, s_f)$ in the approximation theory.

The semantics of the approximation theory $\hat{TF}_{s,e}$ is again defined through a singleton set $\hat{M}_{s,e} = \{(\hat{U}_{s,e}, \hat{I}_{s,e})\}$ of structures. The universe of the approximation theory extends the original universe with a set of integers which are the domain of the precision sort, i.e., $\hat{U}_{s,e} = U_{s,e} \cup \{0, 1, \ldots, n\}$, $\hat{I}_{s,e}(s_p) = \{0, 1, \ldots, n\}$, and $\hat{I}_{s,e}(\omega) = n$. The embedding $h$ is the identity mapping. In order to use precision to regulate the semantics of FP operations, we introduce the notation $(s, e) \downarrow p$ to denote the number of bits in reduced precision $p \in \{0, 1, \ldots, n\}$; more specifically we define

$$(s, e) \downarrow p = \left(3 + \left\lceil (s - 3) \cdot \frac{P}{n} \right\rceil, 3 + \left\lceil (e - 3) \cdot \frac{P}{n} \right\rceil \right),$$

which scales the floating-point sort, however the smallest sort it scales to is $FP_{3,3}$ since smaller well-defined domains contain mostly special values. The approximate semantics of functions is derived from the FP semantics for the reduced bit-widths. For example, $\oplus$ in approximation theory $\hat{TF}_{s,e}$ is defined as

$$\hat{I}_{s,e}((\oplus)(p, r, a, b)) = cast_{s,e}(I_{s,e}(\downarrow p)(\oplus)(r, cast_{s,e}(\downarrow p)(a), cast_{s,e}(\downarrow p)(b)))$$

This definition uses the function $cast_{s,e}$ to map any FP number to a number with $s$ significand bits and $e$ exponent bits, i.e., $cast_{s,e}(a) \in FP_{s,e}$ for any $a \in FP_{s',e'}$.

If $s \geq s'$ and $e \geq e'$ then the casting function does not change the value of the argument, only its sort, i.e., $cast_{s,e}(a) = a$. Otherwise, the cast function performs rounding (if necessary) using a fixed rounding mode. Note that many occurrences of $cast_{s,e}$ can be eliminated in practice, if they only concern intermediate results. For example, consider $\oplus(c_1, \otimes(c_2, a_1, a_2), a_3)$. The result of $\otimes(c_2, a_1, a_2)$ can be directly cast to precision $c_1$ without the need of casting up to full precision when calculating the value of the expression.
4.3 Lifting Constraints to Approximate Constraints

In order to solve a constraint $\phi$ using an approximation theory $\hat{T}$, it is first necessary to lift $\phi$ to an extended constraint $\hat{\phi}$ that includes explicit variables $c_l$ for the precision of each operation. This is done by means of a simple traversal of $\phi$, using a recursive function $L$ that receives a formula (or term) $\phi$ and a position $l \in \mathbb{N}^*$ as argument. For every position $l$, the symbol $c_l$ denotes a fresh variable of the precision sort $\alpha(c_l) = s_p$ and we define

$$L(l, \neg \phi) = \neg L(l, \phi)$$

$$L(l, \phi \circ \psi) = L(l, \phi) \circ L(l, \psi) \quad (\circ \in \{\lor, \land\})$$

$$L(l, x) = x \quad (x \in X)$$

$$L(l, g(t_1, \ldots, t_n)) = g(c_l, L(l, t_1), \ldots, L(l, t_n)) \quad (g \in F \cup P)$$

Then we obtain the lifted formula $\hat{\phi} = L(\epsilon, \phi)$, where $\epsilon$ denotes an empty word. Since $T$-structures can be embedded into $\hat{T}$-structures, it is clear that no models are lost as a result of lifting:

**Lemma 1 (Completeness)** If a $T$-constraint $\phi$ is $T$-satisfiable, then the lifted constraint $\hat{\phi} = L(\epsilon, \phi)$ is $\hat{T}$-satisfiable as well.

In practice, the lifting can make use of expression sharing and cache lifted terms to avoid introduction of unnecessary precision variables or redundant sub-terms.

An approximate model that chooses full precision for all operations induces a model for the original constraint:

**Lemma 2 (Fully precise operations)** Let $\hat{m} = (\hat{U}, \hat{I})$ be a $\hat{T}$-structure, and $\hat{\beta}$ a variable assignment. If $\hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi}$ for an approximate constraint $\hat{\phi} = L(\epsilon, \phi)$, then $m, \beta \models_T \phi$, provided that: 1. there is a $T$-structure $m$ embedded in $\hat{m}$ via $h$, and a variable assignment $\beta$ such that $h(\beta(x)) = \hat{\beta}(x)$ for all variables $x \in \text{fv}(\phi)$, and 2. $\hat{\beta}(c_l) = \hat{I}(\omega)$ for all precision variables $c_l$ introduced by $L$.

The fully precise case however, is not the only case in which an approximate model is easily translated to a precise model. For instance, approximate operations might still yield a precise result for some arguments. Examples of this are constraints in floating-point arithmetic with small integer or fixed-point arithmetic solutions.

A variation of Lemma 2 is obtained by not requiring that all operations are at maximum precision, but that each operation is at a sufficiently high precision, such that it evaluates to the same value as the maximally precise operation in all relevant cases:

**Lemma 3 (Locally precise operations)** Suppose $\hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi}$ for an approximate constraint $\hat{\phi} = L(\epsilon, \phi)$, such that: 1. there is a $T$-structure $m$ embedded in $\hat{m}$ via $h$ and a variable assignment $\beta$ such that $h(\beta(x)) = \hat{\beta}(x)$ for all variables $x \in \text{fv}(\phi)$, and 2. for every sub-expression $g(c_l, \bar{t})$ with $g \in F \cup P$, it holds that $\text{val}_{\hat{m}, \hat{\beta}}(g(c_l, \bar{t})) = \text{val}_{m, \beta}(g(\omega, \bar{t}))$. Then $m, \beta \models_T \phi$.

*Applied to FPA.* Because floating-point numbers of varying bit-widths enjoy the inclusion property, it is easy to see that an approximate model $\hat{m}, \hat{\beta}$ for an approximate $\phi$ which, during model evaluation (validation) does not trigger any rounding decisions, must equally entail the original, precise constraint $\phi$. 
The Approximation Framework

Theorem 1 (Exact evaluation) Let \( \hat{m} \) be the unique element of the singleton set of structures \( \hat{M}_{s,e} \) of theory \( TF_{s,e} \). Suppose \( \hat{m}, \hat{\beta} \models_{TF_{s,e}} \hat{\phi} \) for an approximate constraint \( \hat{\phi} = L(\epsilon, \phi) \), such that: 1. \( m \) is the \( T \)-structure of theory \( TF_{s,e} \) embedded in \( \hat{m} \) via \( h \) (which is the identity function) and \( \beta \) a variable assignment such that \( h(\beta(x)) = \hat{\beta}(x) \) for all variables \( x \in \text{fv}(\hat{\phi}) \), and 2. it is possible to evaluate all operations \( \hat{\phi} \) exactly, i.e. without rounding. Then \( m, \beta \models_{TF_{s,e}} \phi \).

Proof By Lemma 3 and the inclusion property. \( \square \)

Example 1 Lifting the constraints. Consider again the PI controller example given in Section 1.1. Suppose that the program loop is unrolled \( N \) times and translated into single static assignment form, resulting in a set of equations that can be checked for satisfiability. Variables corresponding to the values of program variables at the end of each loop iteration are used as inputs for the next iteration. For the first loop iteration, this leads to the following constraint:

\[
\begin{align*}
\land integral_0 &= 0.0 \quad \text{(initialization)} \\
\land 18.0 &\leq in_0 \leq 22.0 \quad \text{(assumption)} \\
\land error_1 &= set\_point \oplus_{rm} in_0 \quad \text{(line 5)} \\
\land integral_1 &= integral_0 \oplus_{rm} error_1 \quad \text{(line 6)} \\
\land out_1 &= (Kp \oplus_{rm} error_1) \oplus_{rm} (Ki \oplus_{rm} integral_1) \quad \text{(line 7)} \\
&\vdots \\
\land (out_1 < -3.0 \lor out_1 > 3.0 \lor \cdots \lor out_N > 3.0) \quad \text{(violation)}
\end{align*}
\]

where \( Kp, Ki, \) and \( set\_point \) are constant (set to the values given in the PI program, in equations not shown here), and the constant \( rm \) stores the rounding mode. The negated output condition encodes the fact that we search for a violation of the property in any loop iteration.

After lifting those constraints, we obtain the following formula:

\[
\begin{align*}
\land integral_0 &= 0.0 \quad \text{(initialization)} \\
\land 18.0 &\leq^{P_0} in_0 \leq^{P_1} 22.0 \quad \text{(assumption)} \\
\land error_1 &= set\_point \oplus_{rm}^{P_2} in_0 \quad \text{(line 5)} \\
\land integral_1 &= integral_0 \oplus_{rm}^{P_3} error_1 \quad \text{(line 6)} \\
\land out_1 &= (Kp \oplus_{rm}^{P_4} error_1) \oplus_{rm}^{P_6} (Ki \oplus_{rm}^{P_5} integral_1) \quad \text{(line 7)} \\
&\vdots \\
\land (out_1 <^{P_7} -3.0 \lor out_1 >^{P_7} 3.0 \lor \cdots) \quad \text{(violation)}
\end{align*}
\]

The variables \( p_0, p_1, \ldots, p_8, \ldots \) are freshly introduced precision variables of the sort \( s_p \). We use the notation \( \oplus_{rm}^{P_m} \) to express that \( \oplus \) is an operator with four arguments: the precision \( p_2 \), the rounding mode \( rm \), and the two numbers to be added; and similarly for the other operators.
5 Model Refinement Scheme

In the following sections, we will use the approximation framework to successively construct more and more precise solutions of given constraints, until eventually either a genuine solution is found, or the constraints are determined to be unsatisfiable. We fix a partially ordered precision domain \((D_p, \preceq_p)\) (where, as before, \(\preceq_p\) satisfies the ascending chain condition, and has a greatest element), and consider approximation structures \((\hat{U}, \hat{I})\) such that \(\hat{I}(s_p) = D_p\) and \(\hat{I}(\hat{\epsilon}) = \preceq_p\).

Given a lifted constraint \(\hat{\phi} = L(\epsilon, \phi)\), let \(X_p \subseteq X\) be the set of precision variables introduced by the function \(L\). A precision assignment \(\gamma : X_p \rightarrow D_p\) maps the precision variables to precision values. The We write \(\gamma \preceq_p \gamma'\) if for all variables \(c_l \in X_p\) we have \(\gamma(c_l) \preceq_p \gamma'(c_l)\). Precision assignments are partially ordered by \(\preceq_p\). There is a greatest precision assignment \(\gamma_\omega\), which maps each precision variable to \(\omega\). The precision assignment can be obtained from the variable assignment \(\hat{\beta}\) after the solving, but due to its role in controlling the search through the space of approximations (by fixing its values before solving) we separate it from \(\beta\).

The proposed procedure is outlined in Fig. 2. First, an initial precision assignment \(\gamma\) is chosen, depending on the theory \(T\). In Approximate Model Construction, the procedure tries to find \((\hat{m}, \hat{\beta})\), a model of the approximated constraint \(\hat{\phi}\). If \((\hat{m}, \hat{\beta})\) is found, Precise Model Reconstruction tries to translate it to \((m, \beta)\), a model of the original constraint \(\phi\). If this succeeds, the procedure stops and returns the model. Otherwise, Model-guided Approximation Refinement uses \((m, \beta)\) and \((\hat{m}, \hat{\beta})\) to increase the precision assignment \(\gamma\). If Approximate Model Construction cannot find any model \((\hat{m}, \hat{\beta})\), then Proof-guided Approximation Refinement decides how to modify the precision assignment \(\gamma\). If the precision assignment is maximal and cannot be further increased, the procedure has determined unsatisfiability. In the following sections we provide additional details for each of the components of our procedure.

**General properties.** Since \(\preceq_p\) has the ascending chain property, our procedure is guaranteed to terminate and either produce a genuine precise model, or detect unsatisfiability of the constraints. The potential benefits of this approach are that it often takes less time to solve multiple smaller (approximate) problems than to solve the full problem straight away. The candidate models provide useful hints
for the following iterations. The downside is that it might be necessary to solve
the whole problem eventually anyway, which can be the case for unsatisfiable
problems. Whether that is the case depends on the strategy used in the proof-
guided approximation refinement, e.g., maximizing the precision of terms involved
in an unsatisfiable core can cut down the overhead significantly compared to even
increase in precision of all terms. Therefore, our approach is definitely useful when
the goal is to obtain a model, e.g., when searching for counter-examples, but it
can also perform well on unsatisfiable formulas, e.g., when a small unsatisfiable
core can be discovered quickly.

5.1 Approximate Model Construction

Once a precision assignment $\gamma$ has been fixed, existing solvers for the operations
in the approximation theory can be used to construct a model $\hat{m}$ and a variable
assignment $\hat{\beta}$ s.t. $\hat{m}, \hat{\beta} \models \hat{\phi}$. It is necessary that $\hat{\beta}$ and $\gamma$ agree on $X_p$. As an op-
timization, the model search can be formulated in various theory-dependent ways
that provide a heuristic benefit to Precise Model Reconstruction. For example, the
search can prefer models with small values of some error criterion, or to attempt
to find models that are similar to models found in earlier iterations. This can be
done by encoding the problem as an optimization query, assuming one can encode
the desired criteria as part of the formula.

Applied to FPA. Since our FP approximations are again formulated using FP
semantics, any solver for FPA can be used for Approximate Model Construction.
In our implementation, the lifted constraints $\hat{\phi}$ of $\text{TF}_{s,e}$ are encoded in bit-vector
arithmetic, and then bit-blasted and solved using a SAT solver. The encoding of a
particular function or predicate symbol uses the precision argument to determine
the floating-point domain of the interpretation. This kind of approximation reduces
the size of the encoding of each operation, and results in smaller problems handed
over to the SAT solver. An example of theory-specific optimization of the model
search is to prefer models where no rounding occurs during evaluation.

5.2 Reconstructing Precise Models

Alg. 2 provides a high-level sketch for the model reconstruction phase. This al-
gorithm attempts to produce a model $(m, \beta)$ for the original formula $\phi$ from an
approximate model $(\hat{m}, \hat{\beta})$ obtained by solving $\hat{\phi}$. Since we consider arbitrary ap-
xroximations (which might be neither over- nor under-), this translation is non-
trivial; for instance, approximate and precise operations might exhibit different
rounding behavior. In practice, it might still be possible to ‘patch’ approximate
models that are close to real models, avoiding further refinement iterations.

Note that by definition it is possible to embed a $T$-structure $m$ in $\hat{m}$. It is
retrieved, together with the embedding $h$, by $\text{extract}_T$structure in Alg. 2. The
structure $m$ and $h$ will be used to evaluate $\phi$ using values from $\hat{\beta}$. The function
$\text{extract asserted literals}$ determines a set $\text{lits}$ of literals in $\hat{\phi}$ that are true
under $(\hat{m}, \hat{\beta})$, such that the conjunction $\bigwedge \text{lits}$ implies $\hat{\phi}$. For instance, if $\hat{\phi}$ is
Algorithm 2: Model reconstruction

1. $\beta := \emptyset$;
2. $(m, h) := \text{extract}_T\text{structure}(\hat{m})$;
3. $\text{lits} := \text{extract}\_\text{asserted}\_\text{literals}(\hat{m}, \hat{\beta}, \hat{\phi})$;
4. for $l \in \text{lits}$ do
5. \hspace{0.5cm} $(m, \beta) := \text{extend}_\text{model}(l, \beta, h, \hat{\beta}, \hat{m})$;
6. end
7. \text{complete}(\beta, \hat{\beta})$;
8. return $(m, \beta)$;

in CNF, one literal per clause can be selected that is true under $(\hat{m}, \hat{\beta})$. Any pair $(m, \beta)$ that satisfies the literals in $\text{lits}$ will be a $T$-model of $\phi$.

The procedure then iterates over $\text{lits}$, and successively constructs a valuation $\beta : X \rightarrow U$ such that $(m, \beta)$ satisfies all selected literals, and therefore is a model of $\phi$ ($\text{extend}_\text{model}$). During this loop, we assume that $h$ to lift from $m$ to $\hat{m}$, setting all precision variables to greatest precision; formally defined as

$$(\beta \uparrow h)(x) = \begin{cases} \hat{I}(\omega) & \text{if } x \in X_p \\ h(\beta(x)) & \text{otherwise} \end{cases}$$

The precise implementation of $\text{extend}_\text{model}$ is theory-specific. In general, the function first attempts to evaluate a literal $l$ as $\text{val}_{\hat{m}, \hat{\beta} \uparrow h}(l)$. If this fails, the valuation $\beta$ has to be extended, for instance by including values $\hat{\beta}(x)$ for variables $x$ not yet assigned in $\beta$.

After all literals have been successfully asserted, $\beta$ may be incomplete, so we complete it (either randomly or by mapping value assignments from $\hat{\beta}$) and return the model $(m, \beta)$. Note that, if all the asserted literals already have maximum precision assigned then, by Lemma 2, model reconstruction cannot fail.

**Example 2 — Model reconstruction.** In order to illustrate how precise model reconstruction works, recall the formula obtained in Example 1. We fix the number of PI controller loop iterations to $N = 1$, but for reasons of presentation slightly

Applied to FPA. The function $\text{extract}_T\text{structure}$ is trivial for our FPA approximations, since $m$ and $\hat{m}$ coincide for the sort $s_f$ of FP numbers. Further, by approximating FPA using smaller domains of FP numbers, all of which are subsets of the original domain, reconstruction of models is easy in some cases and boils down to padding the obtained values with zero bits. The more difficult cases concern literals with rounding in approximate FP semantics, since a significant error emerges when the literal is re-interpreted using higher-precision FP numbers.

A useful optimization is special treatment of equalities $x = t$ in which one side is a variable $x$ not assigned in $\beta$, and all right-hand side variables are assigned. In this case, the choice $\beta(x) := \text{val}_{\hat{m}, \hat{\beta} \uparrow h}(t)$ will satisfy the equation. Use of this heuristic partly mitigates the negative impact of rounding in approximate FP semantics, since the errors originating in the $(\hat{m}, \hat{\beta})$ will not be present in $(m, \beta)$.

The heuristic is not specific to the floating-point theory, and can be carried over to other theories as well.
change the values of the constants to $K_i = 0.125$, $K_p = 1.25$, and set_point = 3.0. Suppose further that the rounding mode is set to RoundTowardZero, and that the property to be checked is the following: if $2.0 \leq i_0 \leq 4.0$ then $-1.0 \leq o_{t_1} \leq 1.0$. Approximate model construction is performed with the precision assignment $\gamma$ that maps all precision variables $p_0, p_1, \ldots, p_8$ to 0, i.e., all computations are performed in the smallest floating-point domain $FP_{3,3}$.

The columns in Tbl. 2 represent, respectively, the variables in the formula, the terms those variables are assigned, their value in the model of the approximation $\hat{\beta}$ and their value in the reconstructed model $\beta$. The variables in the table are topologically sorted, i.e., their order corresponds to the order of computation in the program, which allows propagation of the rounding error through the formula by interpreting equality as assignment when possible. Before proceeding to model reconstruction, the reader should note that evaluation under the given model $\hat{\beta}$ occurs without rounding, except for the value of $o_{t_1}$, almost meeting the conditions of Lemma 3 and Theorem 1. The exact value of $o_{t_1}$ cannot be represented in $FP_{3,3}$ because $1.375 = 1.011 \cdot 2^0$ which requires 4 significant bits. Since there are only 3 significant bits available, the value is rounded according to the rounding mode $rm$ (bold in Tbl. 2). The given model indeed violates the desired property under $I_{3,3}$. The procedure constructs the model $\beta$, by evaluating the expressions using the interpretation function $I_{53,11}$. Initially, there are no values in $\beta$, so it is populated with values of variables that depend only on constants, cast up to the sort $FP_{53,11}$. Next it proceeds to variables whose value depends on other variables. Since the order is topological, when there are no cycles (like in this example) all the values needed for evaluation are already available in $\beta$. The missing values in $\beta$ are computed by reevaluating the terms assigned to each variable using values of variables already in $\beta$. Since all the variables except $o_{t_1}$ are exact (in the sense that no rounding occurred), then by Lemma 3, their values in $\beta$ and $\hat{\beta}$ are (numerically) equal. In the case of $o_{t_1}$, however, there is a discrepancy between the two values. As there are no cyclic dependencies we can use the more precise value obtained using $I_{53,11}$. In general, the constructed model $\beta$ has to be checked against the constraints, because reconstruction is not guaranteed to succeed. In this example however, the reconstructed $\beta$ is indeed a satisfying assignment for the formula in question.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Defining term</th>
<th>$\hat{\beta}(x)$</th>
<th>$\beta(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_p$</td>
<td>1.25</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>$K_i$</td>
<td>0.125</td>
<td>0.125</td>
<td>0.125</td>
</tr>
<tr>
<td>set_point</td>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>$i_0$</td>
<td></td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>error$_1$</td>
<td>set_point $\oplus_{rm} i_0$</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>integral$_1$</td>
<td>integral$<em>{init} \oplus</em>{rm} error_1$</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>aux$_a$</td>
<td>$K_p \oplus_{rm} error_1$</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>aux$_b$</td>
<td>$K_i \oplus_{rm} error_1$</td>
<td>0.125</td>
<td>0.125</td>
</tr>
<tr>
<td>$o_{t_1}$</td>
<td>aux$<em>a \oplus</em>{rm} aux$_b$</td>
<td>1.25</td>
<td>1.375</td>
</tr>
</tbody>
</table>

Table 2: Model reconstruction from $FP_{3,3}$ to $FP_{53,11}$
5.3 Approximation Refinement

The overall goal of the refinement scheme outlined in Fig. 2 is to find a model of the original constraints using a series of approximations defined by precision assignments $\gamma$. We usually want $\gamma$ to be as small as possible in the partial order of precision assignments, since approximations with lower precision can be solved more efficiently. During refinement, the precision assignment is adjusted so that the approximation of the problem in the next iteration is closer to full semantics. Intuitively, this increase in precision should be kept as small as possible, but as large as necessary. Note that two different refinement procedures are required, depending on whether an approximation is satisfiable or not. We refer to these procedures as Model- and Proof-guided Approximation Refinement, respectively.

5.3.1 Model-guided Approximation Refinement

If a model $(\hat{m}, \hat{\beta})$ of $\hat{\phi}$ is obtained together with a reconstructed model $(m, \beta)$ that does not satisfy $\phi$, we use the procedure described in Alg. 3 for adjusting $\gamma$. Since the model reconstruction failed, there are literals in $\hat{\phi}$ which are critical for $(\hat{m}, \hat{\beta})$, in the sense that they are satisfied by $(\hat{m}, \hat{\beta})$ and required to satisfy $\phi$, but are not satisfied by $(m, \beta)$. Such literals can be identified through evaluation with both $(\hat{m}, \hat{\beta})$ and $(m, \beta)$ (as part of Alg. 3 via `extract_critical_literals`), and can then be traversed, evaluating each sub-term under both structures. If a term $g(c_l, \overline{t})$ is assigned different values in the two models, it witnesses discrepancies between precise and approximate semantics; in this case, an error is computed using the `error` function, mapping to some suitably defined error domain (e.g., the real numbers $\mathbb{R}$ for errors represented numerically). The computed errors are then used to select those operations whose precision argument $c_l$ should be assigned a higher value.

Depending on refinement criteria, the `rank_terms` function can be implemented in different ways. For example, terms can be ordered according to the absolute error which was calculated earlier; if there are too many terms to refine, only a certain number of them will be selected for refinement. An example of a more complex criterion follows:

Error-based selection aims at refining the terms introducing the greatest imprecision first. The absolute error of an expression is determined by the errors of its sub-terms, and the error introduced by approximation of the operation itself. By calculating the ratio between output and input error, refinement tries to select those operations that cause the biggest increase in error. If we assume that theory $T$ is some numerical theory (i.e., it can be mapped to reals in a straightforward manner), then we can define the `error` function (in Alg. 3) as absolute difference between its arguments. Then $\Delta(c_l)$ represents the absolute error of the term $g(c_l, \overline{t})$. This allows us to define the relative error $\delta(c_l)$ of the term $g(c_l, \overline{t})$ as

$$\delta(c_l) = \frac{\Delta(c_l)}{|val_{\hat{m}, \hat{\beta}}(g(\overline{\omega}, \overline{t}))|}.$$ 

Similar measures can be defined for non-numeric theories.

Since a term can have multiple sub-terms, we calculate the average relative input error; alternatively, minimum or maximum input errors could be used. We
Algorithm 3: Model-guided Approximation Refinement

1 \texttt{lits} := \texttt{extract\_critical\_literals}(\hat{m}, \hat{\beta}, \hat{\phi});
2 \textbf{for} \( l \in \texttt{lits} \) \textbf{do}
3 \hspace{1em} \textbf{for} \( g(c_l, \bar{t}) \in \texttt{ordered\_subterms}(l) \) \textbf{do}
4 \hspace{2em} \textbf{if} \( \texttt{val}_{\hat{m}, \hat{\beta}}(g(c_l, \bar{t})) \neq \texttt{val}_{m, \beta T}(g(\omega, \bar{t})) \) \textbf{then}
5 \hspace{3em} \( \Delta(c_l) := \texttt{error}(\texttt{val}_{\hat{m}, \hat{\beta}}(g(c_l, \bar{t})), \texttt{val}_{m, \beta T}(g(\omega, \bar{t}))); \)
6 \hspace{2em} \textbf{end}
7 \hspace{1em} \textbf{end}
8 \textbf{end}
9 \texttt{chosenTerms} := \texttt{rank\_terms}(\Delta);
10 \( \gamma := \texttt{refine}(\gamma, \texttt{chosenTerms}); \)

obtain a function characterizing the increase in error caused by an operation by defining

\[
\text{errInc}(c_l) = \frac{\delta(c_l)}{1 + \frac{1}{k} \sum_{i=1}^{k} \delta(c_{l,i})},
\]

where \( g(c_l, \bar{t}) \) represents the term being ranked. The function \texttt{rank\_terms} then selects terms \( g(c_l, \bar{t}) \) with maximum error increase \( \text{errInc}(c_l) \).

\textit{Applied to FPA.} The only difference to the general case is that we define relative error \( \delta(c_l) \) to be \(+\infty\) if a special value (\( \pm\infty \), NaN) from \( (\hat{m}, \hat{\beta}) \) turns into a normal value under \( (m, \beta) \). Our \texttt{rank\_terms} function ignores terms which have an infinite average relative error of sub-terms. The refinement strategy will prioritize the terms which introduce the largest error, but in the case of special values it will refine the first imprecise terms that are encountered (in bottom up evaluation), because once the special values occur as input error to a term we have no way to estimate its actual error. After ranking the terms using the described criteria, \texttt{rank\_terms} returns the top 30\% highest ranked terms. The precision of chosen terms is increased by a constant value.

\textit{5.3.2 Proof-guided Approximation Refinement}

When no approximate model can be found, some theory solvers may still provide valuable information why the problem could not be satisfied; for instance, proofs of unsatisfiability or unsatisfiable cores. While it may be (computationally) hard to determine which variables absolutely need to be refined in this case (and by how much), in many cases a loose estimate is easy to compute. For instance, a simple solution is to increase the precision of all variables appearing in the literals of an unsatisfiable core.

Given an unsatisfiable formula \( \phi \) in conjunctive normal form (CNF), any unsatisfiable formula \( \psi \) that is a conjunction of a subset of clauses in \( \phi \) is called an unsatisfiable core. If a core \( \psi \) has no proper subformula that is unsatisfiable, it is said to be a minimal unsatisfiable core. Given an unsatisfiable formula \( \psi \) any formula \( \phi \) that contains \( \psi \) is also unsatisfiable, since \( \psi \) is an unsatisfiable core of \( \phi \) in that case. Generalizing this observation to our approximation theory \( \hat{T} \) we get the following lemma:
Algorithm 4: Proof-guided Approximation Refinement

1. \( \psi := \text{extract_unsat_core}(\hat{\phi}, \gamma); \)
2. if \( \forall x \in X \cap \text{vars}(\psi): \gamma(x) = \omega \) then
   return UNSAT;
3. else if \( \phi \in \text{seen_cores} \) then
4. \( \gamma' := \text{refine_everything}(\phi, \gamma); \)
5. return \( \gamma' \);
6. else if \( \exists (m, \beta): m, \beta, \gamma_\omega \models_{TF} \psi \) then
7. \( \text{seen_cores} := \text{seen_cores} \cup \{\psi\}; \)
8. \( \gamma' := \text{refine_everything}(\phi, \gamma); \)
9. return \( \gamma' \);
10. end

Lemma 4 If \( \psi \) is the unsatisfiable core of the lifted formula \( \hat{\phi} \) under precision assignment \( \gamma \) and all precision variables occurring in \( \psi \) have maximal precision, i.e., \( \gamma(x) = \omega \) for all \( x \in X \cap \text{vars}(\psi) \), then formula \( \phi \) is unsatisfiable.

The proof-guided refinement is shown in Alg. 4. Lemma 4 provides a cheap stopping condition for proof-guided refinement. If the found core is at full precision (i.e., was obtained under the exact semantics), then regardless of precision of other constraints the original formula \( \phi \) is guaranteed to be unsatisfiable. However, this is rarely the case (a number of refinement steps is necessary for precision variables to reach value \( \omega \)). Ideally the procedure would get a minimal core \( \psi \) and it would be considerably smaller than the original constraint \( \phi \). In that case, a satisfiability check of \( \psi \) with all the terms at full precision (i.e., \( \omega \)) is likely to be easier than a satisfiability check of \( \phi \). In the case the \( \psi \) is an unsatisfiable core of \( \phi \), this is discovered by solving a considerably smaller formula. If \( \psi \) is not an unsatisfiable core of \( \phi \), then its discovery is due to encoding at small precision, and once encoded at full precision, the search space is going to be expanded enough that the satisfiability check of \( \psi \) is likely to be quick.

In the case that \( \psi \) at full precision is an unsatisfiable core of \( \phi \), proof-guided refinement returns UNSAT (by Lemma 4). Otherwise, we store the formula \( \psi \) in \text{seen_cores}, to be able to skip the satisfiability check if we encounter it (or any of its subsets) in future iterations. All the precision variables are refined, since no useful information is hidden in the core.

If the approximation theory uses a domain with the inclusion property and multiple iterations yield unsatisfiable approximations of the formula \( \hat{\phi} \) then the same solution space is explored repeatedly. Subsequent unsatisfiable iterations are undesirable due to the fact that every previous call is subsumed by the latest one, increasing the solving time unnecessarily. In the case when the approximation theory is FPA, this can be easily avoided by introducing blocking clauses. Between any two iterations, at least one variable had its precision increased, which means that after bit-blasting its encoding will contain additional variables. Since the domain satisfies the inclusion property, that means that all the newly introduced variables implicitly had value \text{false} in the previous iterations. If the approximation of the previous iteration was unsatisfiable, a single clause can be added to prevent revisiting that subspace. The blocking clause expresses that at least one of the newly introduced variables has to be \text{true} (i.e., non-zero).
Example of blocking clauses. Consider the following unsatisfiable formula:

\[ x > y \land x/y < 1 \]

Suppose that in the previous iteration \( x \) and \( y \) were approximated with fixed-point numbers with \( m = 3 \) integral and \( f = 3 \) fractional bits and that the approximation was unsatisfiable. After refinement, the next iteration will use \( m = 5 \) and \( f = 5 \) bits. Below the alignment of the two encodings by the decimal point is shown:

\[
m_2m_1m_0.f_0f_1f_2 \\
m_4m_3m_2m_1m_0.f_0f_1f_2f_3f_4
\]

where \( m_i \) denotes integral bits and \( f_i \) fractional bits, for \( i \in \{0, 1, 2, 3, 4, 5\} \). In the previous iteration, the newly added bits \( f_4, f_3, m_3, m_4 \) implicitly had the value \text{false} (zero). Since the previous satisfiability check returned UNSAT, we can safely exclude those value combinations from the current search. In this example the blocking clause that should be added is

\[ x_{f_4} \lor x_{f_3} \lor x_{m_4} \lor y_{f_4} \lor y_{f_3} \lor y_{m_3} \lor y_{m_4} \]

It evaluates to false when all the newly introduced bits have the values they implicitly had in the previous iteration, preventing further exploration of that part of the search subspace. This technique can be applied to any approximation theory with a domain that exhibits the inclusion property.

6 Experimental Evaluation

To assess the efficacy of our method, we present results of an experimental evaluation obtained through an implementation of the approximation using smaller floating-point numbers (the ‘Smallfloat’ approximation). We implemented this approach as a custom tactic [23] within the Z3 theorem prover [22]. All experiments were performed on Intel Xeon 2.5 GHz machines with a time limit of 1200 sec and a memory limit of 2 GB. The symbols \( T/o \) and \( M/o \) indicate that the time or the memory limit were exceeded.

Implementation details. For the sake of reproducibility of our experiments, we note that our implementation starts with an initial precision mapping \( \gamma \) that limits the precision of all floating-point operations to \( s = 3 \) significand and \( e = 3 \) exponent bits. Upon refinement, operations receive an increase in precision that represents 20% of the width of the full precision. We do not currently implement any sophisticated proof-guided approximation refinement, but our prototype does feature core-based refinement as described in Sec. 5.3.2 and Alg. 4.
Evaluation. Our benchmarks are taken from a recent evaluation of the ACDCL-based MathSAT, by Brain et al. [2]. This benchmark set contains 214 benchmarks, both satisfiable and unsatisfiable ones. The benchmarks originate from verification problems of C programs performing numerical computations, where ranges and error bounds of variables and expressions are verified; other benchmarks are randomly generated systems of inequalities over bounded floating-point variables. We evaluate two versions of our implementation of Smallfloat approximation, one with a simple proof-guided refinement denoted Smallfloat (no cores) and the other featuring core-based proof-guided refinement denoted Smallfloat. We compare against Z3 [22] and MathSAT [6].

The results we obtain are briefly summarized in Tbl. 3, which shows that our method solves more (satisfiable and unsatisfiable) instances than the ordinary bit-blasting-based decision procedure in Z3. Our method solves roughly the same number of satisfiable and unsatisfiable problems as the default procedure based on
bit-blasting in MathSAT, and can handle significantly more satisfiable problems (but fewer unsatisfiable ones) than the ACDCL-based procedure in MathSAT. Few benchmarks are solved by only one solver and they are solved by the best performing solver in their respective category.

Fig. 3–5 provides more detailed results, which show that on satisfiable formulas, our approach (with core-based refinement) is about one order of magnitude faster than Z3, and close to one order of magnitude faster than the default method in MathSAT. In comparison to the ACDCL procedure in MathSAT, the picture is less clear (Fig. 5): while our approximation solves a number of satisfiable problems that
Table 4: Comparison of solver performance on unsatisfiable benchmarks; each entry indicates the number of benchmarks which the approach in the row solves faster than the approach in the column.

<table>
<thead>
<tr>
<th></th>
<th>Z3 (Default)</th>
<th>MathSAT (Default)</th>
<th>MathSAT (ACDCL)</th>
<th>Smallfloat (no cores)</th>
<th>Smallfloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z3 (Default)</td>
<td>-</td>
<td>14</td>
<td>15</td>
<td>59</td>
<td>29</td>
</tr>
<tr>
<td>MathSAT (Default)</td>
<td>56</td>
<td>-</td>
<td>18</td>
<td>64</td>
<td>52</td>
</tr>
<tr>
<td>MathSAT (ACDCL)</td>
<td>73</td>
<td>71</td>
<td>-</td>
<td>75</td>
<td>74</td>
</tr>
<tr>
<td>Smallfloat (no cores)</td>
<td>0</td>
<td>5</td>
<td>12</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Smallfloat</td>
<td>35</td>
<td>18</td>
<td>12</td>
<td>62</td>
<td>-</td>
</tr>
</tbody>
</table>

are hard for MathSAT, it requires more time than MathSAT on other problems. In addition, the ACDCL procedure outperforms all other methods on unsatisfiable problems.

To evaluate the performance of the proof-guided approximation refinement using unsatisfiable cores, we compare all techniques on the unsatisfiable subset of the benchmarks. Tbl. 4 indicates the numbers of benchmarks on which one approach (the row) performs better (solves vs did not solve, or solves faster) than another approach (the column). Both versions of MathSAT perform much better than the other solvers, which is expected. Of particular interest are the two versions of Smallfloat approximation, since they show the impact of core-based refinement on solving. We can see that Smallfloat, featuring core-based refinement, solves 62 benchmarks faster than Smallfloat (no cores), while it is slower on only two instances. This indicates that core-based refinement offers a substantial improvement over the basic proof-guided refinement. Furthermore, by comparing Smallfloat approximation to Z3 (Default), which is the underlying procedure used by both versions of Smallfloat, we can see that it is faster on 37 instances, whereas Smallfloat (no cores) did not outperform Z3 (Default) on any of the benchmarks.

We can conclude that, at least on this benchmark set, the core based refinement offers significant improvement to performance of the approximation framework. It not only improves runtime performance on almost all the benchmarks, it also bridges the gap in performance that is incurred by the approximation framework on more than half of the solved benchmarks.

Overall, it can be observed that our approximation method leads to significant improvements in solver performance, especially where satisfiable formulas are concerned. Our method exhibits complementary performance to the ACDCL procedure in MathSAT; one of the aspects to be investigated in future work is a possible combination of the two methods, using an ACDCL solver to solve the constraints obtained through approximation with our procedure.
7 Conclusion

We present a general method for efficient model construction through the use of approximations. By computing a model of a formula interpreted in suitably approximated semantics, followed by reconstruction of a genuine model in the original semantics, scalability of existing decision procedures is improved for complex background theories. Our method uses a refinement procedure to increase the precision of the approximation on demand. Finally, we show that an instantiation of our framework for floating-point arithmetic shows promising results in practice and often outperforms state-of-the-art solvers.

We plan to further extend the procedure presented here, in particular considering other theories, other approximations, and addressing the case of unsatisfiable constraints. Furthermore, it is possible to solve approximations with different precision assignments in parallel, and use the refinement information from multiple models (or proofs) simultaneously. Increases in precision may then be adjusted based on differences in precision between models, or depending on the runtime required to solve each of the approximations.

Acknowledgments. We would like to thank Alberto Griggio for assistance with MathSAT and help with the benchmarks in our experiments, as well as the anonymous referees for insightful comments.

References

A Additional Experimental Results

The following scatter plots provide additional experimental results, comparing the performance of the ordinary bit-blasting-based decision procedure for FPA in Z3 (without approximation) with the two procedures in MathSAT.

(a) Satisfiable

(b) Unsatisfiable

Fig. 6: MathSAT (ACDCL) vs Z3 Default.

(a) Satisfiable

(b) Unsatisfiable

Fig. 7: Comparison of bit-blasting-based decision procedures.
Paper II
Deciding Bit-Vector Formulas with mcSAT

Aleksandar Zeljić\textsuperscript{1}, Christoph M. Wintersteiger\textsuperscript{2}, and Philipp Rümmer\textsuperscript{1}

\textsuperscript{1} Uppsala University, Sweden
\textsuperscript{2} Microsoft Research

Abstract. The Model-Constructing Satisfiability Calculus (mcSAT) is a recently proposed generalization of propositional DPLL/CDCL for reasoning modulo theories. In contrast to most DPLL(T)-based SMT solvers, which carry out conflict-driven learning only on the propositional level, mcSAT calculi can also synthesise new theory literals during learning, resulting in a simple yet very flexible framework for designing efficient decision procedures. We present an mcSAT calculus for the theory of fixed-size bit-vectors, based on tailor-made conflict-driven learning that exploits both propositional and arithmetic properties of bit-vector operations. Our procedure avoids unnecessary bit-blasting and performs well on problems from domains like software verification, and on constraints over large bit-vectors.

1 Introduction

Fixed-length bit-vectors are one of the most commonly used datatypes in Satisfiability Modulo Theories (SMT), with applications in hardware and software verification, synthesis, scheduling, encoding of combinatorial problems, and many more. Bit-vector solvers are highly efficient, and typically based on some form of SAT encoding, commonly called bit-blasting, in combination with sophisticated methods for upfront simplification. Bit-blasting may be implemented with varying degree of laziness, ranging from eager approaches where the whole formula is translated to propositional logic in one step, to solvers that only translate conjunctions of bit-vector literals at a time (for an overview, see [22]). Despite the huge body of research, aspects of bit-vector solving are still considered challenging, including the combination of bit-vectors with other theories (e.g., arrays or uninterpreted functions), large bit-vector problems that are primarily of arithmetic character (in particular when non-linear), and problems involving very long bit-vectors. A common problem encountered in such cases is excessive memory consumption of solvers, especially for solvers that bit-blast eagerly.

A contrived yet instructive example with very long bit-vectors is given in Fig. 1, adapted from a benchmark of the SMT-LIB QF_BV ‘pspace’ subset [18]. The benchmark tests the overflow behavior of addition. Its model is simple, regardless of the size of bit-vectors $x$ and $y$ ($x$ should consist of only 1-bits, while $y$ should consist of only 0-bits). Finding a model for the formula should in principle be easy, but proves challenging for bit-blasting procedures. Other sources of very long bit-vectors are system memory in hardware verification...
(set-logic QF_BV)
(declare-fun x () (_ BitVec 29980))
(declare-fun y () (_ BitVec 29980))
(assert (and (bvuge x y) (bvule (bvadd x (_ bv1 29980)) y)))

Fig. 1: Simplified example from the ‘pspace’ subset [18] of SMT-LIB, QF_BV

or heap in software verification (e.g., [4]), chemical reaction networks, or gene regulatory networks (e.g., [35]).

Generally, memory consumption is a limiting factor in the application of bit-vector solvers. Increasing the size of bit-vectors that can efficiently be reasoned about would broaden the range of applications, while also simplifying system models for further analysis. With that in mind, we introduce a new model-constrcuting decision procedure for bit-vectors that is lazier than previous solvers. The procedure is defined as an instance of the model-constrcuting satisﬁability calculus (mcSAT [30]), a framework generalizing DPLL and conﬂict-driven learning (CDCL) to non-Boolean domains. Like other SMT solvers for bit-vectors, our procedure is defined on top of well-understood SAT technology; unlike most existing solvers, we treat bit-vectors as ﬁrst-class objects, which enables us to design tailor-made propagation and learning schemes for bit-vector constraints, as well as avoiding bit-blasting of bit-vector operations altogether.

The contributions of this paper are as follows: 1. a novel decision procedure for the theory of bit-vectors that avoids bit-blasting, 2. an extension of the mcSAT calculus to support partial model assignments, 3. a new mcSAT heuristic for generalizing explanations, and 4. an implementation of the procedure and preliminary experimental evaluation of its performance.

1.1 Motivating Examples

We start by illustrating our approach using two examples: a simple bit-vector constraint that illustrates the overall strategy followed by our decision procedure, and a simple family of bit-vector problems on which our procedure outperforms existing bit-blasting-based methods. Consider bit-vectors \( x, y, z \) of length 4, and let \( \oplus \) denote bit-wise exclusive-or and \( \leq_u, <_u \) be unsigned comparison in

\[
\phi \equiv x = y + z \land y <_u z \land (x \leq_u y + y \lor x \oplus z = 0001).
\]

The goal is to ﬁnd an assignment to \( x, y, z \) such that formula evaluates to \( \text{true} \). Fig. 2 illustrates an application of our algorithm to \( \phi \) (after clausiﬁcation). Starting from an empty trail, we assert the unit clauses, denoted by implications with empty antecedents (lines 1 and 2 in Fig. 2a). At this point the procedure chooses between making a model assignment to one of the bit-vector variables, or Boolean decisions. Here, we choose to make a decision and assume \( x \leq_u y + y \) (line 3). Decisions (and model assignments) are denoted with a horizontal line above them in the trail. The Boolean structure of the formula \( \phi \) is now satisfied,
so we search for satisfying model assignments to the bit-vector variables. Here, we decide on $y \mapsto 1111$ (line 4 in Fig. 2a). The literal $y <_u z$ now gives a lower bound for $z$. Our procedure immediately determines that the trail has become infeasible, since no value of $z$ will be consistent with $y = 1111$ and $y <_u z$.

We now need an explanation to restore the trail to a state where it is not infeasible anymore. In mcSAT, an explanation of a conflict is a valid clause with the property that the trail implies falsity of each of the clause literals. One possible explanation in our case is $\neg(y = 1111) \land \neg(y <_u z)$. After resolving the explanation against the trail (in reverse order, similar to Boolean conflict resolution in SAT solvers), at the first point where at least one literal in the conflict clause no longer evaluates to false, the conflict clause becomes an implication and is put on the trail. In this example, as soon as we undo the assignment $y \mapsto 1111$, the literal $\neg(y = 1111)$ can be assumed (line 4 in Fig. 2b). The procedure makes the next legal assignment $y \mapsto 1110$ (line 5 in Fig. 2b). Bounds propagation using $y <_u z$ then implies the model assignment $z \mapsto 1111$ (line 6 in Fig. 2b). Values of $y$ and $z$ imply a unique value 1101 for $x$, however, the model assignment $x \mapsto 1101$ is not legal because it violates $x <_u y + y$ when $y = 1110$. By means of bounds propagation we have detected a conflict in $y = 1110 \land y <_u z \land x = z + y \land x <_u y + y$.

Our procedure tries to generalize conflicts, to avoid re-visiting conflicts of similar shape in the future. Generalization is done by weakening the literals of $y = 1110 \land y <_u z \land x = z + y \land x <_u y + y$, and checking if the conflict persists. First, $y = 1110$ is rewritten to $y \leq_u 1110 \land y \geq_u 1110$; it is then detected that $y \leq_u 1110$ is redundant, because bounds propagation derives unsatisfiability even without it. Now we weaken the literal $y \geq_u 1110$ by changing the constant, say to $y \geq_u 1000$, and verify using bounds propagation that the conflict persists (Example 3). Weakening $y \geq_u 1000$ further would lead to satisfiability. By negation we obtain a valid explanation $\neg(y \geq_u 1000) \lor \neg(y <_u z) \lor \neg(x = z+y) \lor \neg(x <_u y+y)$, which we use to backtrack the trail to a non-conflicted state (line 5 in Fig. 2c). From this point on straight-forward propagation yields a satisfying solution.

After presenting the basic ideas behind our procedure, we argue that it is well suited to problems that stem from model checking applications. Consider
Algorithm 1: Factorial

1 unsigned int factorial = 1u;
2 unsigned int n;
3 for (int i = n; i > 0u; i--){
4     factorial = factorial * i;
5 }
6 assert ( n <= 1 || factorial % 2u == 0u)

the simple C program shown in Alg. 1. The program computes the factorial of some value \( n \) by multiplying the factors starting from \( n \) and counting down. We add an assertion at the end, which checks whether \( \text{factorial} \) is even if the value of \( n \) is greater than one. We use bounded model checking and unwind the loop a fixed number of iterations, to generate formulas of increasing complexity. Fig. 3 shows the performance of \texttt{mcBV} (our prototype) and state-of-the-art solvers on these benchmarks (Boolector [10] is the winner of the QF_BV track of the 2015 SMT competition; Z3 [29] is our baseline as \texttt{mcBV} uses the Z3 parser and preprocessor). On this class of benchmarks, \texttt{mcBV} performs significantly better than Z3 and comparably to Boolector.

1.2 Related Work

The most popular approach to solving bit-vector formulas is to translate them to propositional logic and further into conjunctive normal form via the Tseitin translation [33] (bit-blasting), such that an off-the-shelf SAT solver can be used to determine satisfiability. In contrast, our approach does not bit-blast, and we attempt to determine satisfiability directly on the word level. Our technique builds on the Model-Constructing Satisfiability Calculus recently developed by Jovanović and de Moura [30, 23]. Our approach is similar in spirit to previous work by Bardin et al. [1], which avoids bit-blasting by encoding bit-vector problems into integer arithmetic, such that a (customized) CLP solver for finite do-

![Fig. 3: Factorial example performance](image-url)
mains can be used. A different angle is taken by Wille et al. [34] in their SWORD tool, which uses vector-level information to increase the performance of the SAT solver by abstracting some sub-formulas into ‘modules’ that are handled similar to custom propagators in a CLP solver.

On the one hand, various decision problems involving bit-vectors have recently been shown to be of fairly high complexity [24, 19, 25] and on the other hand, some fragments are known to be decidable in polynomial time; for instance, Bruttomesso and Sharygina describe an efficient decision procedure for the ‘core’ theory of bit-vectors [11], based on earlier work by Cyrluk et al. [15], who defined this fragment via syntactic restriction to extraction and concatenation being the only bit-vector operators that are permitted. There is also a small body of work on the extension of decision procedures for bit-vectors that do not have a fixed size. For instance, Bjørner and Pichora [7] describe a unification-based calculus for (non-)fixed size bit-vectors, while Möller and Ruess [28] describe a procedure for (non-)fixed size bit-vectors that bit-blasts lazily (‘splitting on demand’).

Most SMT solvers implement lazy and/or eager bit-blasting procedures. These either directly, eagerly translate to a Boolean CNF and then run a SAT solver, or, especially when theory combination is required, they use a lazy bit-blast that translates relevant parts of formulas on demand. This is the case, for instance in Boolector [10], MathSAT [13], CVC4 [2], STP [20], Yices [17], and Z3 [29]. Hadarean et al. [22] present a comparison and evaluation of eager, lazy, and combined bit-vector solvers in CVC4.

Griggio proposes an efficient procedure for the construction of Craig interpolants of bit-vector formulas ([27] via translation to QF_LIA, quantifier-free linear integer arithmetic [21]). Interpolants do have applications in mcBV, e.g., for conflict generalization, but we do not currently employ such methods.

Model checkers that do not use SMT solvers sometimes implement their own bit-blasting procedures and then use SAT solvers, BDD-based, or AIG-based solvers. This is often done in bounded model checking [5, 6], but more recently also in IC3 [9], Impact [27], or k-induction [32]. Examples thereof include CBMC [14], EMBM [26], NuSMV2 [12]. In some cases model-checkers based on abstract interpretation principles use bit-vector solvers for counter-example generation when the proof fails; this is the case, for instance, for the separation-logic based memory analyzer SLAyer [3, 4].

In recent times bit-vector constraints are also used for formal systems analysis procedures in areas other than verification, for instance in computational biology [35] where Dunn et al. [16] identify and analyze a bit-vector model for pluripotent stem-cells via an encoding of the model into bit-vector constraints. Similarly, bit-vector solvers are used within interactive and automated theorem provers to construct bit-vector proofs, for instance, in Isabelle/HOL [8].

2 Preliminaries: Bit-Vector Constraints

We consider a logic of quantifier-free fixed-width bit-vector constraints, defined by the following grammar, in which $\phi$ ranges over formulas and $t$ over bit-vector
terms:
\[
\phi \ ::= \ true \ | \ false \ | \ p \ | \ \neg \phi \ | \ \phi \land \phi \ | \ \phi \lor \phi \ | \ t \bullet t
\]
\[
t \ ::= \ 0_n \ | \ 1_n \ | \ \cdots \ | \ (0|1)^+ \ | \ x \ | \ \text{extract}_p^n(t) \ | \ ! | \ t \circ t
\]

Here, \( p \) ranges over propositional variables; expressions \( 0_n, 1_n, \ldots \) are decimal bit-vector literals of width \( n \); literals \( (0|1)^+ \) represent bit-vectors in binary; \( x \) ranges over bit-vector variables of size \( \alpha(x) \); predicates \( \bullet \in \{=, \leq_s, \leq_u\} \) represent equality, signed inequality (2's complement format), and unsigned inequality, respectively; the operator \( \text{extract}_p^n \) represents extraction of \( n \) bits starting from position \( p \) (where the left-most bit has position 0); \( ! \) is bit-wise negation; binary operators \( \circ \in \{+ , \times, \div, \ |, \ +, \ll, \gg, \sim\} \) represent addition, multiplication, (unsigned) integer division, bit-wise and, bit-wise or, bit-wise exclusive or, left-shift, right-shift, and concatenation, respectively. We assume typing and semantics of bit-vector constraints are defined as usual.

An atom is a formula \( \phi \) that does not contain \( \neg, \land, \lor \). An atom is flat if it is of the form \( x \leq_s y, x \leq_u y, \) or \( x = t, \) and \( t \) does not contain nested operators. A literal is an atom or its negation. A clause is a disjunction of literals. When checking satisfiability of a bit-vector constraint, we generally assume that the constraint is given in the form of a set of clauses containing only flat atoms.

### 3 mcSAT with Projections

We now introduce the framework used by our decision procedure for bit-vector constraints, based on a generalized version of mcSAT [30]. In contrast to previous formulations of mcSAT, we include the possibility to partially assign values to variables; this enables assignments that only affect some of the bits in a bit-vector, which helps us to define more flexible propagation operators. mcSAT with projections is first defined in general terms, and tailored to the setting of bit-vector constraints in the subsequent sections, resulting in mcBV.

We define our framework in the form of a transition system, following the tradition of DPLL(T) [31] and mcSAT [30]. The states of the system have the form \( \langle M, C \rangle \), where \( M \) is the trail (a finite sequence of trail elements), and \( C \) is a set of clauses. The trail \( M \) consists of: (1) decided literals \( l \), (2) propagated literals \( e \rightarrow l \), and (3) model assignments \( \pi(x) \mapsto \alpha \). A literal \( l \) (either decided or propagated) is considered to be true in the current state if it appears in \( M \), which is denoted by \( l \in M \). Model assignments \( \pi(x) \mapsto \alpha \) denote a partial assignment of a value \( \alpha \) to a variable \( x \), where \( \pi \) is a projection function.

We consider constraints formulated over a set of types \( \{T_1, T_2, \ldots, T_n\} \) with fixed domains \( \{T_1, T_2, \ldots, T_n\} \), and a finite family \( \{\pi_i\}_{i \in I} \) of surjective functions \( \pi_i : T \rightarrow T' \) (here called projections) between the domains. Types can for instance be bit-vector sorts of various lengths. A partial model assignment \( \pi(x) \mapsto \alpha \) with projection \( \pi : T \rightarrow T' \) expresses that variable \( x \) of type \( T \) is assigned some value \( \beta \in T \) such that \( \pi(\beta) = \alpha \), where \( \alpha \in T' \). A trail can contain multiple partial assignments to the same variable \( x \); we define the partial domain of a
variable \( x \) under a trail \( M \) as

\[
\text{Domain}(x, M) = \bigcap_{(\pi(x) \rightarrow \alpha) \in M} \{ \beta \in T \mid \pi(\beta) = \alpha \}.
\]

We call a trail \( M \) assignment consistent if the partial assignments to variables are mutually consistent, i.e., the partial domain \( \text{Domain}(x, M) \) of each variable \( x \) is non-empty. If the partial domain of a variable \( x \) contains exactly one element, i.e., \( \text{Domain}(x, M) = \{ \beta \} \), then we say that all the partial assignments to \( x \) in \( M \) form a full model assignment; in the original mcSAT calculus, this is denoted by \( x \rightarrow \beta \). Assignment consistency is violated if \( \text{Domain}(x, M) = \emptyset \). We generally require that projections \( \{ \pi_i \}_{i \in I} \) are chosen in such a way that assignment consistency and full assignments can be detected effectively for any trail \( M \). In addition, projections are required to be complete in the sense that every full model assignment \( x \rightarrow \beta \) can be expressed as some finite combination of partial assignments \( \{ \pi_j(x) \rightarrow \alpha_j \}_{j \in J} \). More formally, for every \( \beta \in T \) there exists a finite set of partial model assignments \( S \) such that \( \text{Domain}(x, S) = \{ \beta \} \). Inclusion of the identity function among projections enables expression of full model assignments directly.

Given a trail \( M \), an interpretation \( v[M] = \{ x_1 \rightarrow \beta_1, x_2 \rightarrow \beta_2, \ldots, x_k \rightarrow \beta_k \} \) is constructed by collecting all full model assignments \( x_i \rightarrow \beta_i \) implied by \( M \). The value \( v[M](t) \) of a term or formula \( t \) is its value under the interpretation \( v \), provided that all variables occurring in \( t \) are interpreted by \( v \); or \( \text{undef} \) otherwise. We define a trail extension \( \tilde{M} \) of a trail \( M \) as any trail \( \tilde{M} = [M, M_0] \) such that \( M_0 \) consists only of (partial) model assignments to variables already appearing in \( M \), and furthermore each variable \( x \) that appears in \( M \) has a unique value assigned from its partial domain; \( \text{Domain}(x, M) \neq \emptyset \) implies that \( |\text{Domain}(x, \tilde{M})| = 1 \). This ensures that assignment consistency is maintained.

Evaluation of literals in respect to the trail \( M \) is achieved using a pair of functions \( \text{value}_B \) and \( \text{value}_T \), defined as

\[
\text{value}_B(l, M) = \begin{cases} 
\text{true} & l \in M \\
\text{false} & \neg l \in M \\
\text{undef} & \text{otherwise}
\end{cases}
\quad \text{and} \quad
\text{value}_T(l, M) = v[M](l).
\]

A trail \( M \) is consistent if it is assignment consistent, and for all literals \( l \in M \) it holds that \( \text{value}_T(l, M) \neq \text{false} \). A trail \( M \) is said to be complete if it is consistent and every literal \( l \) on the trail \( M \) can be evaluated in the theory, i.e. \( \text{value}_T(l, M) = \text{true} \). A trail which has no complete extensions is called infeasible. Note that if a trail is inconsistent then it is also infeasible.

The value of a literal in a consistent state (consistent trail) is defined as

\[
\text{value}(l, M) = \begin{cases} 
\text{value}_B(l, M) & \text{value}_B(l, M) \neq \text{undef} \\
\text{value}_T(l, M) & \text{otherwise}
\end{cases},
\]

which is extended to clauses in the obvious way.
Propagate
\[ (M, C) \rightarrow ([M, c \rightarrow l], c) \text{ if } \forall i : value(l_i, M) = false \]
\[ value(l, M) = undef \]

Decide
\[ (M, C) \rightarrow ([M, l], C) \text{ if } l \in B, value(l, M) = undef \]

Conflict
\[ (M, C) \rightarrow (M, C) \vdash e \text{ if } c \in C, value(c, M) = false \]

Sat
\[ (M, C) \rightarrow sat \text{ if } M \text{ is complete} \]
\[ value(c, M) = true \text{ for all } c \in C \]

Forget
\[ (M, C) \rightarrow (M, C \setminus \{c\}) \text{ if } c \in C \text{ is a learned clause} \]

**Fig. 4: Clausal search rules**

**Evaluation strength.** We remark that there is some freedom in the way \( value_T \) is defined: even if \( v[M](l) = undef \) for some literal \( l \) (because \( l \) contains variables with undefined value), the trail \( M \) might still uniquely determine the value of \( l \).

In general, our calculus can use any definition of \( value^*_T \) that satisfies

1. \( v[M](l) \neq undef \) implies \( value^*_T(l, M) = v[M](l) \), and
2. \( value^*_T(l, M) \neq undef \) implies that for every extension \( \tilde{M} \) of \( M \) it holds that \( value_T(l, \tilde{M}) = value^*_T(l, M) \).

These properties leave room for a trade-off between the strength of reasoning and computational effort invested to discover such implications. For example, suppose that a bit-vector variable \( x \) of length 3, under trail \( M \) has the partial domain \( \text{Domain}(x, M) = \{000, 001, 010\} \). For a literal \( l = (x < 100) \), evaluation yields \( value_T(l, M) = undef \). It is easy to see that \( value_T(l, \tilde{M}) = true \) in every trail extension \( \tilde{M} \) of \( M \), however, so that a lazier mode of evaluation could determine \( value^*_T(l, M) = true \).

With a more liberal evaluation strategy, propagations and conflicts are detected earlier, though perhaps at higher cost.

### 3.1 A Calculus with Projections

The transitions of our calculus are mostly the same as those of mcSAT [30]. The clausal rules are shown in Fig. 4 and Fig. 5. Theory rules are presented in Fig. 6. The only significant modification of the mcSAT calculus is the T-Decide rule, which we define in terms of partial assignments and partial domains. As in mcSAT, it is assumed that a finite basis \( B \) of literals is given, representing all literals that are taken into account in decisions, propagations, or when constructing conflict clauses and explanations. \( B \) at least has to contain all atoms, and the negation of atoms occurring in the clause set \( C \). The function \( \text{explain} \) is supposed to compute explanations of infeasible trails \( M \) (which correspond to theory lemmas in DPLL(T) terminology). An explanation of \( M \) is a clause \( e \) such that 1. \( e \)
4. Searching for Models with mcBV

Resolve

\[ (\{M, d \to l\}, C) \vdash c \longrightarrow (M, C) \vdash r \quad \text{if } \neg l \in c \]
\[ r = \text{resolve}(c, d, l) \]

Consume

\[ (\{M, d \to l\}, C) \vdash c \longrightarrow (M, C) \vdash c \quad \text{if } \neg l \in c \]
\[ (\{M, l\}, C) \vdash c \longrightarrow (M, C) \vdash c \quad \text{if } \neg l \in c \]

Backjump

\[ (\{M, N\}, C) \vdash c \longrightarrow (\{M, c \to l\}, C) \quad \text{if } c = l_1 \lor \ldots \lor l_m \lor l \]
\[ \forall i : \text{value}(L_i, M) = \text{false} \]
\[ \text{value}(l, M) = \text{undef} \]
\[ N \text{ starts with a decision} \]

Unsat

\[ (M, C) \vdash \text{false} \longrightarrow \text{unsat} \]

Learn

\[ (M, C) \vdash c \longrightarrow (M, C \cup \{c\}) \vdash c \quad \text{if } c \notin C \]

Fig. 5: Clausal conflict analysis rules

is valid; 2. all literals \( l \in e \) evaluate to false on \( M \) (i.e., \( \text{value}(l, M) = \text{false} \)); 3. all literals \( l \in e \) occur in the basis \( B \).

In order to state correctness of the calculus, we need one further assumption about the well-foundedness of partial assignments: for every sequence of partial assignments to a variable \( x \), of the form \([\pi_1(x) \mapsto \alpha_1, \pi_2(x) \mapsto \alpha_2, \ldots] \), we assume that the sequence of partial prefix domains

\[
\text{Domain}(x, []) \\
\text{Domain}(x, [\pi_1(x) \mapsto \alpha_1]) \\
\text{Domain}(x, [\pi_1(x) \mapsto \alpha_1, \pi_2(x) \mapsto \alpha_2]) \\
\ldots
\]

eventually becomes constant. This ensures that partial assignment of a variable cannot be refined indefinitely. Correctness of mcSAT with projections is then be proven in largely the same manner as in mcSAT:

**Theorem 1 (Correctness [30]).** Any derivation starting from the initial state \([\{\}, C]\) eventually terminates in state sat, if \( C \) is satisfiable, or in state unsat, if \( C \) is unsatisfiable.

4 Searching for Models with mcBV

We now describe how the mcSAT calculus with projections is tailored to the theory of bit-vectors, leading to our procedure mcBV. The theory of bit-vectors already contains a natural choice for the projections, namely the extract functions, of which we use a finite subset as projections. To ensure completeness of this subset (in the sense that every full model assignment has a representation as a combination of partial model assignments), we include all one-bit projections \( \pi^k = \text{extract}_i^1 \), selecting the \( i \)-th bit of a bit-vector of length \( k \).
T-Propagate

\[\langle M, C \rangle \quad \rightarrow \quad \langle [M, e \rightarrow l], C \rangle \quad \text{if } l \in B, \text{ value}(l, M) = \text{undef} \]

T-Decide

\[\langle M, C \rangle \quad \rightarrow \quad \langle [M, \pi(x) \leftrightarrow \alpha], C \rangle \quad \text{if } x \text{ is a (theory) variable in } C \]

\[\text{Domain}(x, M) \neq \text{Domain}(x, [M, \pi(x) \leftrightarrow \alpha]) \]

\[M, \pi(x) \leftrightarrow \alpha \text{ is consistent} \]

T-Conflict

\[\langle M, C \rangle \quad \rightarrow \quad \langle M, C \models e \rangle \quad \text{if } M \text{ is infeasible} \]

\[e = \text{explain}(M) \]

T-Backjump-Decide

\[\langle [M, \pi(x) \leftrightarrow \alpha], N], C \models e \quad \rightarrow \quad \langle [M, l], C \rangle \quad \text{if } \exists i : \text{value}(l_i, M) = \text{undef} \]

\[\text{value}(l, M) = \text{undef} \]

T-Consume

\[\langle [M, \pi(x) \leftrightarrow \alpha], C \rangle \models c \quad \rightarrow \quad \langle M, C \models c \rangle \quad \text{if } \text{value}(c, M) = \text{false} \]

Fig. 6: Theory search and conflict rules

In practice, our prototype implementation maintains a trail \(M\) as part of its state, and attempts to extend the trail with literals and model assignments such that the trail stays consistent, every literal on the trail eventually becomes justified by a model assignment (i.e., \(\text{value}_T(l, M) = \text{true}\) for every literal \(l\) in \(M\)), and every clause in \(C\) is eventually satisfied. A conflict is detected if either some clause in \(C\) is found to be falsified by the chosen trail elements (which is due to literals or model assignments), or if infeasibility of the trail is detected.

Since the calculus is model constructing, there is a strong preference to justify all literals on the trail through model assignments, i.e., to make the trail complete, before making further Boolean decisions. Partial model assignments are instrumental for this strategy: they enable flexible implementation of propagation rules that extract as much information from trail literals as possible. For instance, if the trail contains the equation \(x = \text{extract}_2^3(y)\) and a model assignment \(x \rightarrow 101\), propagation infers and puts a further partial assignment \(\text{extract}_2^3(y) \rightarrow 101\) on the trail, by means of the T-Decide rule. This partial assignment is subsequently used to derive further information about other variables. For this, we defined bit-precise propagation rules for all operators; our solver includes native implementations of those rules and does not have to resort to explicit bit-blasting. Similarly to Boolean Constraint Propagation (BCP), propagation on the level of bit-vectors is often able to detect inconsistency of trails (in particular variables with empty partial domain) very efficiently.

Once all possible bit-vector propagations have been carried out, but the trail \(M\) is still not complete, the values of further variables have to be decided upon through T-Decide. In order to avoid wrong decisions and obvious conflicts, our implementation also maintains over-approximations of the set of feasible values of each variable, in the form of bit-patterns and arithmetic intervals. These sets are updated whenever new elements are pushed on the trail, and refined
using BCP-equivalent propagation and interval constraint propagation (ICP). Besides indicating values of variables consistent with the trail, these sets offer cheap infeasibility detection (when they become empty), which is crucial for the T-Propagate and T-Conflict rules. Also, frequently one of these sets becomes singleton, in which case a unique model assignment for the variable is implied.

4.1 Efficient Representation of Partial Model Assignments

Our procedure efficiently maintains information about the partial domains of variables by tracking bit-patterns, which are strings over the 3-letter alphabet \{0, 1, u\}; the symbol u represents undefined bits (don’t-cares). We say that bit-vector \(x\) matches bit-pattern \(p\) (both of length \(k\)) iff \(x\) is included in the set of vectors covered in the bit-pattern; formally we define

\[
\text{matches}(x, p) = \bigwedge_{0 \leq i < k} x_i = p_i ,
\]

where \(x_i\) and \(p_i\) denote \(i\)-th bit of \(x\) and \(p\). The atom \(\text{matches}(x, p)\) is not a formula in the sense of our language of bit-vector constraints, but for the sake of presentation we treat it as such in this section.

For long bit-vectors, representation of partial domains using simple bit-patterns can be inefficient, since linear space is needed in the length of the bit-vector variable. To offset this, we use run-length encoding (RLE) to store bit-patterns. Besides memory compression, RLE speeds up propagation, as it is not necessary to process every individual bit of a bit-vector separately. The complexity then depends on the number of bit alternations, as shown in the following example demonstrating exclusive-or evaluation on both representations.

**Example 1.** Each digit in the output represents one bit operation, in standard bit-vectors (left) and run-length encoded bit-vectors (right):

\[
\begin{align*}
x & \quad 0000111111 \quad & x & \quad 0^3 0^2 1^3 1^2 \\
y & \quad 1100000111 \quad & y & \quad 1^3 0^2 0^3 1^2 \\
x \oplus y & \quad 1110011100 \quad & x \oplus y & \quad 1^3 0^2 1^3 0^2
\end{align*}
\]

4.2 Maintaining Partial Domain Over-approximations

To capture arithmetic properties and enable efficient propagation, our implementation stores bounds \(x \in [x_l, x_u]\) for each variable \(x\). The bounds are updated when new elements occur on the trail, and bounds propagation is used to refine bounds. Note that bit-patterns and arithmetic bounds abstractions sometimes also refine each other. For example, lower and upper bounds are derived from a bit-pattern, by replacing all \(u\) bits with 0 and 1 respectively. Conversely, if the lower and upper bound share a prefix, then they imply a bit-pattern with the same prefix and the remaining bits set to \(u\).
5 Conflicts and Explanations

Explanations are the vehicle used by our calculus to generalize from conflicts. Given an infeasible trail $M$, an explanation $\text{explain}(M)$ is defined to be a valid clause $E = l_1 \lor \cdots \lor l_n$ over the finite basis $B$, such that every literal $l_i$ evaluates to $\text{false}$ under the current trail, i.e., $\text{value}(l_i, M) = \text{false}$ for every $i \in \{1, \ldots, n\}$. Explanations encode contradictory assumptions made on the trail, and are needed in the T-Consumes and T-Conflict rules to control conflict resolution and backtracking, as well as in T-Propagate to justify literals added to the trail as the result of theory propagation.

Since the trail $M$ is inconsistent at the beginning of conflict resolution, it is always possible to find explanations that are simply disjunctions of negated trail literals; to this end, every propagated literal $c \rightarrow l$ is identified with $l$, and every model assignment $\pi(x) \rightarrow \alpha$ as the formula $\pi(x) = \alpha$.

5.1 Greedy generalization

We present a greedy algorithm for creating explanations that abstract from concrete causes of conflict. To this end, we assume that we have already derived some correct (but not very general) explanation $e = \neg t_1 \lor \neg t_2 \lor \cdots \lor \neg t_n \lor \neg b_1 \lor \neg b_2 \lor \cdots \lor \neg b_m$, where $t_1, \ldots, t_n$ denote literals with $\text{value}_T(t_i, M) = \text{true}$, and $b_1, \ldots, b_m$ literals with $\text{value}_T(t_i, M) = \text{undef}$ but $\text{value}_B(b_i, M) = \text{true}$. The former kind of literal holds as a result of model assignments, whereas the latter literals occur on the trail either as decisions or as the result of propagation. The key observation is that the literals $t_1, \ldots, t_n$ allow over-approximations (replacements with logically weaker literals), as long as the validity of the overall explanation clause is maintained, in this way producing a more general explanation.

Our procedure requires the following components as input (apart from $e$):

- for each literal $t_i$ (for $i \in \{1, \ldots, n\}$), a finite lattice $(T_i, \Rightarrow)$ of conjunctions of literals $T_i \subseteq \{l_1 \land \cdots \land l_k \mid l_1, \ldots, l_k \in B\}$ ordered by logical implication, with join $\sqcup_i$ and meet $\sqcap_i$, and the property that $t_i \in T_i$. The set $T_i$ provides constraints that are considered as relaxation of $t_i$.
- a heuristic satisfiability checker $\text{hsat}$ to determine the satisfiability of a conjunction of literals. The checker $\text{hsat}$ is required to (1) be be sound (i.e., $\text{hsat}(\phi) = \text{false}$ implies that $\phi$ is actually unsatisfiable), (2) to correctly report the validity of $e$,

$$\text{hsat}(t_1 \land \cdots \land t_n \land b_1 \land \cdots \land b_m) = \text{false},$$

and (3) to be monotonic in the following sense: for all elements $l, l' \in T_i$ with $l \Rightarrow l'$ in one of the lattices, and for all conjunctions $\phi$, $\psi$ of literals, if $\text{hsat}(\phi \land l' \land \psi) = \text{false}$ then $\text{hsat}(\phi \land l \land \psi) = \text{false}$.
Algorithm 2: Explanation relaxation

**Input:** Raw explanation $\bigvee_{i=1}^{n} \neg t_i \vee \bigvee_{i=1}^{m} \neg b_i$; lattices $(T_i, \Rightarrow)_{i=1}^{n}$; satisfiability checker $\text{hsat}$.

**Output:** Refined explanation $\bigvee_{i=1}^{n} \neg t_i^f \vee \bigvee_{i=1}^{m} \neg b_i$.

1. $\phi_n \leftarrow b_1 \land \cdots \land b_m$;
2. **for** $i \leftarrow 1$ **to** $n$ **do**
   3. $t_i^a \leftarrow t_i$;
   4. $B_i \leftarrow \emptyset$;
5. **end**

6. changed $\leftarrow$ true;
7. **while** changed **do**
   8. changed $\leftarrow$ false;
   9. **for** $i \leftarrow 1$ **to** $n$ **do**
      10. if $\exists t \in T_i \setminus \{t_i^a\}$ with $t_i^a \Rightarrow t$ and $\forall l \in B_i. t \neq l$ then
          11. if $\text{hsat}(\bigwedge_{j=1}^{i-1} t_j^a \land t \land \bigwedge_{j=i+1}^{n} t_j^a \land \phi_b)$ then
              12. $B_i \leftarrow B_i \cup \{t\}$;
          else
              13. $t_i^a \leftarrow t$;
              14. changed $\leftarrow$ true;
          **end**
      **end**
   **end**
19. **return** $\neg t_1^a \lor \cdots \lor \neg t_n^a \lor \neg b_1 \lor \cdots \lor \neg b_m$;

The pseudo-code of the procedure is shown in Alg. 2, and consists mainly of a fixed-point loop in which the literals $t_1, \ldots, t_n$ are iteratively weakened, until no further changes are possible (lines 6–19). The algorithm keeps blocking sets $B_i$ of conjunctions of literals (for $i \in \{1, \ldots, n\}$) that have been considered as relaxation for $t_i$, but were found to be too weak to maintain a valid explanation.

**Lemma 1.** Provided a correct explanation clause as input, Alg. 2 terminates and produces a correct refined explanation.

The next two sections describe two instances of our procedure: one targeting explanations that are primarily of arithmetic character, and one for explanations that mainly involve bit-wise operations.

### 5.2 Greedy Bit-wise Generalization

If the literals of an explanation clause are primarily bit-wise in nature, the relaxation considered in our method is to weaken the bit-patterns associated with the variables occurring in the conflict. For every literal $t_i$ of the form $\text{matches}(x, p)$,
where \( x \) is a bit-vector variable and \( p \) a bit-pattern of width \( k = \alpha(x) \) implied by the trail, we choose the lattice \((T_i, \Rightarrow)\) with

\[
T_i = \{\text{false}\} \cup \{\text{matches}(x, a) \mid a \in \{0, 1, u\}^k\}.
\]

This set contains a constraint that is equivalent to \textit{true}, namely the literal \textit{matches}(\(x, u^k\)). Conflicts involving (partial) assignments allow us to start near the bottom of the lattice and we weaken the literal as much as possible in order to cover as many similar assignments as possible. Concretely, the weakening is performed by replacing occurrences of 1 or 0 in the bit-pattern by \( u \).

Our prototype satisfiability checker \textit{hsat} for this type of constraints is implemented by propagation (Sect. 4), which is able to show validity of raw explanation clauses, and similarly handles bit-pattern relaxations. Our implementation covers all operations, but it is imprecise for some arithmetic operations.

\begin{example}
Consider the trail

\[
M = [\ldots, y \mapsto 0^4, x \mapsto 1^10^1, \text{extract}^2_2(y) = x].
\]

This trail is inconsistent because literal \textit{extract}^2_2(y) = x evaluates to false in the theory. A simple explanation clause is \( \neg\text{matches}(y, 0^4) \lor \neg\text{matches}(x, 1^10^1) \lor \neg(\text{extract}^2_2(y) = x) \). The generalization procedure now tries to generalize this naive explanation by weakening the literals. Fig. 7 shows steps of generalizing the conflict observed in trail \( M \). One by one, bits in the pattern are set to \( u \) and it is checked whether the new clause is valid. If it is valid, the new bit-pattern is altered further, otherwise we discard it and continue with the last successfully weakened pattern. Note that we are not restricted to changes to only one bit, or even only one literal at a time.
\end{example}

### 5.3 Greedy Arithmetic Generalization

If the literals of an explanation clause are primarily arithmetic, the relaxation considered in our method is to replace equations (that stem from model assignments on the trail) with inequalities or interval constraints: for every literal \( t_i \) of the form \( x = v_k \), where \( x \) is a bit-vector variable and \( v_k \) a literal bit-vector constant of width \( k = \alpha(x) \), we choose the lattice \((T_i, \Rightarrow)\) with

\[
T_i = \{\text{false}\} \cup \{x = a \mid a \in \{0, 1\}^k\} \cup \\
\{a \leq_u x \land x \leq_u b \mid a, b \in \{0, 1\}^k, a < b\}
\]
6. Experiments and Evaluation

This set contains a constraint that is equivalent to \( \text{true} \ (0^k \leq_u x \wedge x \leq_u 1^k) \), and similarly constraints that only impose concrete lower or upper bounds on \( x \), and equalities \( (x = v_k) \in T_i \).

Our satisfiability checker \( \text{hsat} \) for this type of constraints implements interval constraint propagation, covering all bit-vector operations, although it tends to yield more precise results for arithmetic than for bit-wise operations. If ICP shows that an interval becomes empty, then the generalization succeeds because the conflict persists. Otherwise, generalization fails when ICP reaches a fix point or exceeds a fixed number of steps (to avoid problems with slow convergence).

Example 3. For readability purposes we switch to numerical notation in this example. Recall the basic explanation of the trail conflict shown in Fig. 2b in the motivating example (Sect. 1.1):

\[
\neg(y = 14) \lor \neg(y <_u z) \lor \neg(x = z + y) \lor (x <_u y + y)
\]

We rewrite the first negated equality as a disjunction of negated inequalities. Then the iterative generalization procedure starts. For each bound literal, the procedure first attempts to remove it (by weakening it to a \( \text{true} \)-equivalent in the lattice \( T_i \)). If unsuccessful, it navigates the lattice of literals using binary search over bounds. Fig. 8 shows the steps in this particular example.

6 Experiments and Evaluation

To evaluate the performance of our mcBV implementation we conducted experiments on the SMT-LIB QF_BV benchmark set, using our implementation of mcBV in F#, on a Windows HPC cluster of Intel Xeon machines. The benchmark set contains 49971 files in SMT2 format, each of which contains a set of assertions and a single (check-sat) command. The timeout for all experiments is at 1200 sec and the memory limit is 2 GB.

Currently we do not implement any advanced heuristics for clause learning, clause deletion, or restarts and thus mcBV does not outperform any other solver consistently. We present a runtime comparison of mcBV with the state-of-the-art SMT solvers Boolector and Z3 on a selected subset of the QF_BV benchmarks see Fig. 9. On the whole benchmark set, mcBV is not yet competitive with Boolector or Z3, but it is interesting to note that mcBV performs well on some of the benchmarks in the ‘sage’, ‘sage2’ and ‘pspace’ sets, as well as the entirety of the
Fig. 9: Runtime comparison on selected subsets of SMT QF_BV. Markers for ‘sage’ and ‘sage2’ are smaller to avoid clutter; #< shows the number of benchmarks that only mcBV solves or mcBV solves quicker than both Z3 and Boolector.

‘brummayerbieere4’ set. Those sets contain a substantial number of benchmarks that mcBV could solve, but Z3 and Boolector cannot. The ‘pspace’ benchmarks are hard for all solvers as they contains very large bit-vectors (in the order of 20k bits) which will often result in the bit-blaster running out of memory; this is reflected in the small clusters at the bottom right in Fig. 9. The table in Fig. 9 gives the number of instances solved by each approach. While our prototype performs relatively well on selected subsets, it will need improvements and advanced heuristics to compete with the state-of-the-art on all of QF_BV.

7 Conclusion

We presented a new decision procedure of the theory of bit-vectors, which is based on an extension of the Model-Constructing Satisfiability Calculus (mcSAT). In contrast to state-of-the-art solvers, our procedure avoids unnecessary bit-blasting. Although our implementation is prototypical and lacks most of the more advanced heuristics used in solvers (e.g., variable selection/decision heuristics, lemma learning, restarts, deletion strategies), our approach shows promising performance, and is comparable with the best available solvers on a number of benchmarks. This constitutes a proof of concept for instantiation of the mcSAT framework for a new theory; we expect significantly improved performance as we further optimise our implementation. Additionally, we improve the flexibility of the mcSAT framework by introducing projection functions and partial assignments, which we believe to be crucial for the model-constructing approach for bit-vectors.
References

Recent licentiate theses from the Department of Information Technology

2016-009  Timofey Mukha: Inflow Generation for Scale-Resolving Simulations of Turbulent Boundary Layers
2016-008  Simon Sticko: Towards Higher Order Immersed Finite Elements for the Wave Equation
2016-007  Volkan Cambazoglou: Protocol, Mobility and Adversary Models for the Verification of Security
2016-006  Anton Axelsson: Context: The Abstract Term for the Concrete
2016-005  Ida Bodin: Cognitive Work Analysis in Practice: Adaptation to Project Scope and Industrial Context
2016-004  Kasun Hewage: Towards a Secure Synchronous Communication Architecture for Low-power Wireless Networks
2016-002  Rubén Cubo: Mathematical Modeling for Optimization of Deep Brain Stimulation
2016-001  Victor Shcherbakov: Radial Basis Function Methods for Pricing Multi-Asset Options
2015-006  Hanna Holmgren: Towards Accurate Modeling of Moving Contact Lines
2015-005  Siyang Wang: Analysis of Boundary and Interface Closures for Finite Difference Methods for the Wave Equation
2015-004  Pavol Bauer: Parallelism and Efficiency in Discrete-Event Simulation