UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1603

From Machine Arithmetic to Approximations and back again

Improved SMT Methods for Numeric Data Types

ALEKSANDAR ZELJIĆ

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017

ISSN 1651-6214
ISBN 978-91-513-0162-4
urn:nbn:se:uu:diva-334565

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, Lägerhyddsvägen 2, Uppsala, Tuesday, 23 January 2018 at 09:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Prof. Dr. Armin Biere (Johannes Keppler University, Linz, Austria).

Abstract

Zeljić, A. 2017. From Machine Arithmetic to Approximations and back again. Improved SMT Methods for Numeric Data Types. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1603. 55 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0162-4.

Safety-critical systems, especially those found in avionics and automotive industries, rely on machine arithmetic to perform their tasks: integer arithmetic, fixed-point arithmetic or floating-point arithmetic (FPA). Machine arithmetic exhibits subtle differences in behavior compared to the ideal mathematical arithmetic, due to fixed-size representation in memory. Failure of safety-critical systems is unacceptable, due to high-stakes involving human lives or huge amounts of money, time and effort. By formally proving properties of systems, we can be assured that they meet safety requirements. However, to prove such properties it is necessary to reason about machine arithmetic. SMT techniques for machine arithmetic are lacking scalability. This thesis presents approaches that augment or complement existing SMT techniques for machine arithmetic.

In this thesis, we explore approximations as a means of augmenting existing decision procedures. A general approximation refinement framework is presented, along with its implementation called UppSAT. The framework solves a sequence of approximations. Initially very crude, these approximations are fairly easy to solve. Results of solving approximate constraints are used to either reconstruct a solution of original constraints, obtain a proof of unsatisfiability or to refine the approximation. The framework preserves soundness, completeness, and termination of the underlying decision procedure, guaranteeing that eventually, either a solution is found or a proof that solution does not exist. We evaluate the impact of approximations implemented in the UppSAT framework on the state-of-the-art in SMT for floating-point arithmetic.

A novel method to reason about the theory of fixed-width bit-vectors called mcBV is presented. It is an instantiation of the model constructing satisfiability calculus, mcSAT, and uses a new lazy representation of bit-vectors that allows both bit- and word-level reasoning. It uses a greedy explanation generalization mechanism capable of more general learning compared to traditional approaches. Evaluation of mcBV shows that it can outperform bit-blasting on several classes of problems.

Keywords: SMT, Model construction, Approximations, floating-point arithmetic, machine arithmetic, bit-vectors

Aleksandar Zeljić, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Aleksandar Zeljić 2017

ISSN 1651-6214
ISBN 978-91-513-0162-4
urn:nbn:se:uu:diva-334565 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-334565)

To Ana Marija

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Deciding Bit-Vector Formulas Using mcSAT —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer.
Theory and Applications of Satisfiability Testing - 19th International Conference, SAT 2016. Proceedings.

II An Approximation Framework for Decision Procedures —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer.
Journal of Automated Reasoning 58(1): 127-147 (2017)

Extended version of:
Approximations for Model Construction —
Aleksandar Zeljić, Christoph M. Wintersteiger, Philipp Rümmer.
Automated Reasoning — 7th International Joint Conference, IJCAR 2014. Proceedings. **Best Paper Award**.

III Exploring Approximations for Floating-Point Arithmetic using UppSAT —
Aleksandar Zeljić, Peter Backeman, Christoph M. Wintersteiger, Philipp Rümmer.
Technical report, published electronically in arXiv.
arXiv:1711.08859v2

Reprints were made with permission from the publishers.

Comments on Paper Contributions

Paper I

I am the main author of Paper I as well as the main developer of mcBV solver presented therein.

Paper II

I am the main author of Paper II, both the conference and the extended journal version. I am the developer of the implementation presented in Paper II. The conference paper is is completely subsumed by the journal publication and is not reprinted in this thesis.

Paper III

The work presented in Paper III is joint work with Peter Backeman. We have made equal contributions to the implementation of UppSAT. The authorship of the paper is shared between us. Peter took the lead in conducting the experimental evaluation and I took the lead in writing the paper.

Acknowledgments

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

Philipp, I owe you thanks beyond what I can put into words. I can no longer imagine what my life would have looked like had I not taken this opportunity. Doing this PhD allowed me to grow both professionally and personally, and has presented me with opportunities I would never have dreamed of. Thank you!

Christoph, thank you for all the discussions, tips and advice. My implementation skills have gotten a long way thanks to you!

I also want to thank my colleagues who have made working at the IT department interesting and fun. Special thanks go to the members of the Real-Time Systems Group, current and past, 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.

Contents

1 Introduction 11
1.1 Research questions 12

2 A Motivating Example 14
2.1 Analyzing Program Execution 14
2.2 Choosing the Reasoning Domain 19

3 Machine Arithmetic 22
3.1 Bounded Integer Arithmetic 23
3.2 Fixed-Point Arithmetic 24
3.3 Floating-Point Arithmetic 25

4 A Brief Introduction to Mathematical Logic 27
4.1 Propositional Logic 27
4.2 Predicate Logic 29

5 The Satisfiability Problem 31
5.1 The SAT problem 31
5.1.1 The Davis-Putnam Procedure 32
5.1.2 The DPLL Procedure 32
5.1.3 The CDCL Procedure 35
5.2 The SMT Problem 38
5.2.1 Beyond DPLL(T) 39

6 Reasoning about Machine Arithmetic 41
6.1 Decision Procedures for Bit-Vector Arithmetic 41
6.2 Reasoning about Floating-Point Arithmetic 43
6.2.1 Decision Procedures for Floating-Point Arithmetic 43
6.2.2 Other Approaches to Reasoning about FPA 46

7 Contributions 47

8 Conclusion 49

9 Sammanfattning på Svenska 50

References 52

1. Introduction

Computer systems permeate modern society: government, transportation, communication, economy, even households rely upon dozens of them. In an ideal world, these systems function as intended, but in reality, some of them will not. In many cases, the consequences of failure are not severe. After all, being unable to access email or social networks for a while is frustrating at worst. Some systems, however, perform a critical role in a larger system, such as control or ensuring safety. Failure of such systems can cost time, money, or even human lives.

Some more well-known examples of include:

- the Pentium FDIV bug [21] — In 1994, Intel's Pentium processor occasionally returned incorrect results of floating-point division. The scope of the impact of this bug is unclear, as it affected personal computers, however replacing the chips cost Intel hundreds of millions of dollars.
- the Ariane 5 accident [51] — In 1996, during the maiden flight of rocket Ariane 5, a 64-bit floating-point value was assigned to a 16-bit integer. As a consequence, the value changed magnitudes and resulted in a crash of one of the navigation components, and caused the rocket's destruction less than a minute after launch. The error occurred due to reuse of some sub-systems from Ariane 4, which resulted in a corruption of data critical for flight navigation.
- the Patriot missile defense bug [8] — In 1991, a Patriot missile failed to intercept an incoming missile and hit a barracks killing 28 people and injuring many more. The failure was due to an incorrect trajectory calculation, caused by a rounding error. The exact increment of time could not be represented precisely, so it was rounded off. Multiplied by a sufficiently large value the round-off error was significant enough for the missile to miss the interception.
- the Java's sorting algorithm bug [29] — In 2015, Java's default sorting algorithm was shown to be incorrect in certain corner cases. The bug went unnoticed for years and affected a wide array of users, from mobile devices to cloud services.

Considerable effort is invested to avoid such accidents. However, ensuring correct functioning of computer systems is not easy. Computer systems consist of hardware and software working in concert and they need to function correctly, both independently and combined. As the above examples illustrate, sources of errors are many: error in hardware design was behind the FDIV

bug, lack of awareness of numerical properties of rounding caused the Patriot missile failure, while the Ariane 5 accident was caused by a mistake in overall system design. Instead of ensuring correctness of the entire system, which is often infeasible due to size and complexity, the focus is on sub-systems with safety-critical roles, such as control or safety. Failure of those systems can result in devastating consequences.

How can we ensure correctness of software? Intended behavior of a system is described by a *specification*. If the implementation exhibits behavior that is not in line with the specification we consider it faulty or incorrect. Testing is a technique widely used in the development process that detects faulty behavior, however, it cannot guarantee its absence. More and more, production of computer systems is using formal methods to ensure that the final product is error-free. Formal methods refer to techniques that are based on formal mathematics. They are used in specification, development, and verification, among others. Formal specification removes ambiguities present in natural language and subjects it to rigorous examination before providing a formal description of the system. In development, formal methods are used to synthesize an implementation directly from the specification. Verification aims to prove that the implementation satisfies the specification. To achieve this, we need: 1. a formal language (e.g., first-order logic in conjunction with one or more theories), 2. a program logic (e.g., a Hoare-style calculus), and 3. a reasoning procedure for the chosen formal setting.

Consider, for instance, an airplane flight controller. It relies on various parameters and arithmetic computation to adjust vector and speed of the aircraft. Typically, this involves using *floating-point arithmetic* (FPA) to perform computations over real-valued data. Floating-point arithmetic is an approximation of the real arithmetic and exhibits some unintuitive and subtle behavior, from the perspective of the real arithmetic. Reasoning about machine arithmetic, i.e. integer and floating-point arithmetic, is necessary to verify the correctness of software such as a flight controller. It 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 one or more background theories. For example, theories relevant for reasoning about machine arithmetic are the theory of bit-vectors and the theory of floating point arithmetic. One of the main challenges for reasoning about machine arithmetic is scalability. This thesis tackles the problem of scalability of procedures for reasoning about machine arithmetic.

1.1 Research questions

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

Question 1: Can use of approximations improve performance and scalability of the existing decision procedures?

Question 2: Many SMT solvers reason about machine arithmetic by reducing it to propositional satisfiability. Can a decision procedure based on native domain reasoning outperform reduction-based approaches?

Question 3: Can approximations be combined with existing procedures in a general manner, independently of particular solver implementations?

Question 1 explores whether existing procedures can be improved through the use of non-conservative approximations. In recent years, conservative approximations have proven to be a useful tool to tackle the ever-increasing problem size. By abstracting away parts of the problem, the solver can focus on essential aspects of the problem. Typically approximation refinement schemes rely on conservative properties of approximations. We investigated a systematic way to use non-conservative approximations to solve constraints. In particular, the framework approximates floating-point constraints using reduced precision floating-point arithmetic. The framework and evaluation of experimental results are presented in Paper I.

Question 2 explores alternatives to a well-established approach to reasoning about machine arithmetic. Namely, many solvers encode machine arithmetic into a less expressive theory. For instance, floating-point arithmetic is reduced to bit-vectors, whereas bit-vectors are reduced to propositional logic. This approach allows the use of existing procedures to solve the problem. However, the downside is that the resulting formula can become too large to even begin solving. Furthermore, some structural information is lost in the encoding. For example, the fact that certain bit represents sign or parity, or that a collection of operations represents an addition operation has no meaning 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.

Question 3 stems from the affirmative answer to Question 1. Reduced-precision floating-point approximation, presented in Paper I, improves the scalability of bit-blasting. The approximation and the approximation refinement framework are implemented as a tactic within the Z3 [30] SMT solver. We re-implement the approximation refinement framework as UppSAT, an abstract approximating SMT solver. UppSAT takes an SMT solver as a back-end and yields an approximating version of the solver. Furthermore, different procedures might benefit from different kinds of approximations. UppSAT defines templates that allow easy and modular specification of approximations. Paper III presents how to specify approximations in UppSAT. Additionally it compares performance of several instances of UppSAT obtained by combining different approximations and back-end solvers.

2. A Motivating Example

We start with an example which will illustrate the necessary steps to reason about program behavior. Algorithm 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 Algorithm 1, the computation of $P_n(x)$ is performed iterative. 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 an integer counter variable `i`. The higher order derivatives cycle through values $1, 0, -1, 0$. The iterations when the value of a higher-order derivative is zero are 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 `EPS` is achieved.

When considering the code in Algorithm 1, we might wonder whether 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, $sin(x) = sin(-x)$ for all x. These properties hold for the polynomial P_n and we can show that mathematically. The real question, however, is whether the implementation `sine_expansion` satisfies these properties. Is $|\texttt{sine\_expansion}(x)| \leq 1$ for all floating-point values x? And is $\texttt{sine\_expansion}(x) = \texttt{sine\_expansion}(-x)$ for all floating-point values x? In order to discover answers to these questions, we need to analyze the program execution in some way.

2.1 Analyzing Program Execution

Methods to analyze program executions of interest to this thesis are *deductive* and *model-checking-based* methods. In both cases, there is a specification

```
1  #include <math.h>
2
3  #define EPS 1E-20
4
5  double sine_expansion(double x){
6    double x_sq = x * x;
7    double ratio = x;
8    double sum = ratio;
9    int i;
10   for (i = 3; fabs(ratio) > EPS; i+=2) {
11     ratio *= x_sq/(i*(i-1));
12
13     if(i%4 == 1)
14       sum += ratio;
15     else
16       sum -= ratio;
17   }
18
19   return sum;
20 }
```

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

and a property to be verified. Deductive methods rely on mathematical proof systems to show that the property holds. 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. Some of these tools allow the user to provide a crucial piece of information to guide the proof, e.g., by supplying a critical statement or a lemma. Given a finite-state transition system, model-checking tests whether the property holds in every possible execution of the system. The property is usually given as a formula in some temporal logic, such as CTL [19]. The system is shown to be a model of the property by systematic exploration of the state space and explicit check in each state whether the property holds. Model-checking has the advantage of being fully automatic, but suffers from the *state space explosion problem*. A version of *model checking* [19] that tests only traces of particular length is called *bounded model checking* [6].

We apply bounded model checking (BMC) to the program in Figure 1. A program is interpreted as a state transition system. A state is represented by the combination of values assigned to the program variables. Executing a program statement results in a transition to a new state, based on the changes to the variables. The transition system is encoded into a formula $\psi_{program}$,

which describes the semantics of the program. This is straightforward for programs that contain no loops or recursion. Handling loops, however, requires unrolling the loop a fixed number of times, say n, and then constructing the formula. By fixing the number of loop iterations, the property is checked only for a particular class of program executions. Let formula $\psi_{property}$ encode the property that we want to check. To check whether the property is satisfied for all executions, we test whether there are any executions that violate the property. This amounts to a query to an SMT solver whether

$$\psi_{program} \wedge \neg \psi_{property}$$

is satisfiable. If this formula is satisfiable, then for some combination of inputs, the program can reach a state 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 $\psi_{property}$. If there is no model, then 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 since it only considers executions of a particular length.

Encoding Programs as Formulas
When considering how to encode program executions as formulas, there are two aspects that attract attention. The first is that programs are sequential (i.e., statement order is important), whereas formulas are unordered. The other is that program variables are mutable, i.e. they can hold different values during an execution, while mathematical variables are immutable. To handle these two properties, a sequence of mathematical variables is introduced 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 $i_{init}, i_0, i_1, \ldots$ Every time an assignment to a variable occurs, the variable is indexed with a new label. This corresponds to the *single static assignment form* (SSA) [25], which requires that each variable is assigned only once during the program execution. Since the example contains only initialization and the loop, input arguments are labeled with *in*, variables at the time of initialization are labelled by *init*, otherwise the value of the counter variable i at the time of assignment is used as a label. 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, only the initialization of variables occurs.

Let us consider the control flow of execution in Figure 2.1 (left). Since the body of the loop is not executed, the loop condition `fabs(ratio) > EPS` is not satisfied. That will be the case whenever the input value $\mathtt{x}_{in}$ is smaller or equal than the predefined value `EPS`.

```
x_sq_init = x_in * x_in;                x_sq_init = x_in * x_in ∧
ratio_init = x_in;                      ratio_init = x_in ∧
sum_init = ratio_init;                  sum_init = ratio_init ∧
i_init = 0;                             i_init = 0 ∧
i_0 = 3;                                i_0 = 3 ∧
{!(fabs(ratio_init) > EPS))}            ¬ (fabs(ratio_init) > EPS))
return sum_init;
```

Figure 2.1. This figure shows: the execution of Algorithm 1 with no loop unrollings (left) and the corresponding first-order formula $\phi_0(x_{in})$ (right).

We proceed to construct a series of formulas $\phi_i(x_{in})$ where i denotes number of loop unrollings. The formula with no unrollings, denoted $\phi_0(x_{in})$, is shown in Figure 2.1 (right). It is simple to construct, because each variable is assigned only once.

```
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 Figure 1 with one loop iteration being executed.

Figure 2.2 shows a program execution in which the loop body is executed exactly once. It models all executions in which the initial condition of the `for`-loop is met, but is violated after only one execution of its body. The loop is unwound for various values of n to obtain sequences of executions to analyze.

By unwinding the loop, the program takes on a shape that is easy to analyze. During the unwinding, the additional conditions, such as the met or failed loop conditions need to be taken into account. These can be 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 [5], 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.

```
1  (set-logic QF_FPBV)
2
3  (declare-fun x_in () (_ FloatingPoint 11 53))
4  (declare-fun x_sq_init () (_ FloatingPoint 11 53))
5  (declare-fun ratio_init () (_ FloatingPoint 11 53))
6  (declare-fun sum_init () (_ FloatingPoint 11 53))
7  (declare-fun i_init () (_ BitVec 32))
8  (declare-fun i_0 () (_ BitVec 32))
9
10 (assert (= x_sq_init (fp.mul RTZ x_in x_in)))
11 (assert (= ratio_init x_in ))
12 (assert (= sum_init ratio_init))
13 (assert (= i_init (_ bv0 32)))
14 (assert (= i_0 (_ bv3 32)))
15 (assert (not
16         (fp.gt (fp.abs ratio_init)
17                (fp (_ bv0 1)
18                    (_ bv989 11)
19                    (_ bv3233525618163131 52)))))
20
21 (assert (not
22         (fp.leq (fp.abs sum_init)
23                 (fp (_ bv0 1)
24                     (_ bv1023 11)
25                     (_ bv0 52)))))
26
27 (check-sat)
```

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

Example 2.1.2. Given the program execution in the SSA form, shown in Figure 2.1, we produce the corresponding SMT-LIB formula, shown in Figure 2.3. An SMT-LIB file consists of several parts: 1. preamble — which declares meta information, such as the logic being used, the author(s) of the file, comments, status of the formula, etc. 2. function declarations — denoted by keyword **declare-fun** followed by the function name, sorts of its arguments (empty

brackets denote zero arguments, i.e. a constant symbol) and finally the sort of the symbol itself, 3. the formula — represented by one or more **assert** statements, and 4. call to check for satisfiability — in the form of the **check-sat** keyword.

In Figure 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 of the formula is declared. In this case, it is the quantifier-free logic of floating-point arithmetic and fixed-width bit-vectors (or `QF_FPBV`). It contains the `FloatingPoint`, `RoundingMode` and `BitVector` sorts, constants for the five rounding modes and all the standard floating-point and bit-vector operations. Normally, 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 $\oplus, \ominus, \odot$ are denoted as fp.add, fp.sub, fp.mul in `QF_FPBV`. The lines 3-9 declare the constants corresponding to the program variables and their sorts. For example, on line 3, x_{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, `(_ bv3 32)` denotes a bit-vector literal of bit-width 32 representing integer value 3. Lines 3-8 correspond to program declarations (and in some cases initializations). Lines 10-19 describe the semantics of the zero-length unfolding of the 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, $\neg(sum \leq 1)$, is asserted on line 21. And finally, **check-sat** instructs the solver to check the satisfiability of the formula.

2.2 Choosing the Reasoning Domain

In the preceding example, the program is encoded as a first-order formula of the theory of floating-point arithmetic and bit-vectors. 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 the floating-point arithmetic is merely a vehicle of computation. Each of these problems has its own flavor of semantics. We

can reason about the real arithmetic, the floating-point arithmetic, the fixed-point arithmetic, or the bounded integer arithmetic. SMT-LIB logics enable expressing statements about values, variables and operations in these domains as formulas.

Algorithms which determine whether a formula is a theorem of a particular theory are called *decision procedures*. We briefly introduce some of the decision procedures relevant for these arithmetic domains:

- SMT-style decision procedures for floating-point arithmetic can be divided into two categories: *eager* and *lazy*. The eager approach looks at the IEEE-754 standard-compliant hardware and encodes it into bit-vector arithmetic which is reduced to propositional satisfiability (SAT) by a method called *bit-blasting* [48]. 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 is a combination of two encodings: from FPA to bit-vector arithmetic (BVA) and from BVA to propositional satisfiability (SAT), which can result in very large formulas. An example of the lazy approach is the ACDCL algorithm [11]. Instead of explicitly representing the floating-point values, they can be abstracted as intervals (i.e., pairs of bounds $(lower, upper)$). 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 [55]. Semi-automated approaches using interactive theorem provers also exist. For example, the Coq proof assistant [33] can be used to formally reason about floating-point arithmetic [54].
- 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 [48]. Procedures for non-linear arithmetic exist and they, unsurprisingly, have a higher complexity than procedures for linear arithmetic. 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 a considerably simpler way to represent real values than floating-point arithmetic and also easier to reason about. It can be encoded into theory of bit-vectors in a straightforward manner, since it is 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.

3. Machine Arithmetic

Most number systems used in computers are positional. In a positional number system a number is stored as a string of digits, and position of a digit determines its contribution to the overall value. The set of digits available depends on the *base b* of the number system, also called the *radix*. The digits in base b 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. In 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) representation can be obtained from binary by grouping adjacent binary digits into triples and mapping them to an octal digit. Similarly, hexadecimal (base 16) representation is obtained by grouping 4 adjacent binary digits. This allows a compact representation of binary literals that would be too long to write out in binary.

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:

$$d = \sum_{i=-p}^{n} d_i \cdot b^i \tag{3.1}$$

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 base of the number system:

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

This representation is not unique, but a normal form can be imposed upon it. For example, enforcing the condition that $0 < |significand| < base$ 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 $exponent = 0$, but other values could be used if we are dealing with a particular range of 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 comparison.

Representing numerical data in a computer faces some (mathematically) unexpected behavior. This is usually the consequence of the fact that memory is finite, so we limit ourselves to a certain size in memory. The most obvious consequences are overflows and rounding error but also some more subtle effects can 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 no fractional digits. In addition, practical implementation imposes another constraint, the fixed size in memory, which limits the number of *significand* digits. Most hardware offers support for bounded integer arithmetic. Unbounded integers have to be implemented in software, at the expense of performance.

Given a fixed bit-width l the sets of values that can be represented are:

$$\boldsymbol{D}_{unsigned} = \{0, 1, 2, \ldots, 2^l - 1\}$$

and

$$\boldsymbol{D}_{signed} = \{-2^{l-1}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{l-1} - 1\}$$

The actual encoding of negative values can be done in a number of ways, with the two's complement being the most common approach. When a result cannot be stored using l bits, the bits that cannot be stored are said to *overflow*. This phenomenon leads to a number of problems, e.g., the program might crash or produce a seemingly incorrect result, e.g., by adding two large positive numbers resulting in a small or a negative number. Beside the standard arithmetic operations, such as $+, -, \cdot, /$, frequently used are arithmetic and logical shifts, bit-wise operations etc. The integer data-type is inherently arithmetic, but sees various non-standard operations being used for efficiency or utility reasons. This usage gave rise to the perspective of bit-vectors, binary strings that are stored in memory and can be manipulated in various ways. There are several approaches to formally reason about integer arithmetic. The SMT-LIB theory of fixed size bit-vectors is the most straightforward, since it models actual hardware implementation of integer operations. An alternative is to encode bounded integer arithmetic into linear integer arithmetic.

3.2 Fixed-Point Arithmetic

Fixed-point arithmetic is equivalent to scientific notation in which 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. The density of real values represented by fixed-point arithmetic is uniform. Typically, fixed-point arithmetic is 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 the fixed length, there is a trade-off between precision and magnitude of the values that can be represented. If the total bit-width is fixed, 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 representation. These operations require additional shifting to align the results with the 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. The rounding is usually towards zero, but other rounding modes can be implemented.

3.3 Floating-Point Arithmetic

Intuitively, floating-point arithmetic, corresponds to a scientific notation with the condition that $0 \leq |\mathit{significand}| < \mathit{base}$. Additionally, the significand and the exponent are subjected to a fixed number of digits. The floating-point arithmetic includes several special values: signed zeros, signed infinities, and special not-a-number values. The remainder of this section presents the floating-point arithmetic in more detail.

The floating-point arithmetic is defined in the IEEE-754 standard [44]. It is a systematic approximation of the 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 $+\inf$ and $-\inf$. 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 *NaN*s). 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 *NaN*s, which store diagnostic information as part of its encoding. The IEEE-754 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 floating-point numbers with s significand bits and e exponent bits by $FP_{s,e}$:

$$\left\{ (-1)^{sgn} \cdot sig \cdot 2^{exp-s} \middle| \begin{array}{l} sgn \in \{0,1\}, \\ sig \in \{0,\ldots,2^s-1\}, \\ exp \in \{-2^{e-1}+3,\ldots,2^{e-1}\} \end{array} \right\} \cup \left\{ \begin{array}{ll} NaN, & +\infty, \\ -\infty, & -0 \end{array} \right\}$$

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 $\odot$ denote its counterpart in the theory of floating-point arithmetic. A floating-point operation using rounding mode *rm* over arguments *a* and *b* (in prefix notation) is defined as:

$$(\odot\ rm\ a\ b) = (round(rm, (\cdot\ a\ b))).$$

Since FPA is an approximation of real arithmetic, whenever rounding occurs, a rounding error is introduced in the computation. Rounding also affects some basic mathematical properties of operations, such as the associativity property 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 a number) the significand represents the same number of different values. As a consequence, the representable values are denser in smaller magnitudes, and sparser in greater magnitudes. Varying density combined with rounding leads to different magnitudes of errors during computation, depending on the values involved. The values of exponents involved determine the rounding error of a given operation, which is expressed in *units-in-last-place*, or *ULP*. ULPs are particularly relevant for computation of round-off errors and efforts to produce error bounds. Another peculiar interaction is that addition of a non-zero floating-point value and a value of much greater magnitude can appear as a zero-addition. As a consequence, big difference in magnitude of values can affect the 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 correctly can be tricky, since the special values usually propagate through computations. If the programmer loses sight of this fact and produces code that ignores the emergence of special values, the entire computation could be compromised. Automated methods to detect when the FPA semantics leads the implementation to deviate from the expected outcomes would be very valuable tools for development of safety-critical software. The techniques explored in this thesis could prove crucial for practical applicability of such tools.

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 Σ consists of:

- constant literals: $\top$ (*true*) and $\bot$(*false*),
- a countable set of propositional variables P,
- logical connectives: $\neg$ (*negation*), $\wedge$ (*conjunction*), $\vee$ (*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 Σ such that:

- constant literals $\top$ and $\bot$ are propositional formulas
- propositional variables are propositional formulas
- if A and B are propositional formulas then so are: $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$.

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 $\neg, \wedge, \vee$ 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 $val_v(\bot) = 0$
- $val_v(x) = v(p),\ p \in P$
- $val_v(\neg A) = \begin{cases} 1, \text{ if } val_v(A) = 0 \\ 0, \text{ if } val_v(A) = 1 \end{cases}$
- $val_v(A \wedge B) = \begin{cases} 1, \text{ if } val_v(A) = 1 \text{ and } val_v(B) = 1 \\ 0, \text{ otherwise} \end{cases}$
- $val_v(A \vee B) = \begin{cases} 0, \text{ if } val_v(A) = 0 \text{ and } val_v(B) = 0 \\ 1, \text{ otherwise} \end{cases}$
- $val_v(\phi \Rightarrow B) = \begin{cases} 0, \text{ if } val_v(A) = 1 \text{ and } val_v(B) = 0 \\ 1, \text{ otherwise} \end{cases}$
- $val_v(A \Leftrightarrow B) = \begin{cases} 1, \text{ if } val_v(A) = val_v(B) \\ 0, \text{ otherwise} \end{cases}$

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 exists.

Example 4.1.1. Formula $p \Rightarrow p$ is a tautology, $p \wedge \neg p$ is a contradiction and $p \Rightarrow q$ 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., [42]).

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 $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- 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 α.

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 α to assign sorts also to variables.

Given a multi-sorted signature Σ 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 argument sorts 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 argument sorts 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 ϕ and ψ are formulas then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$. If x is a variable, s a sort, and ϕ a formula then, $\forall x : s.\phi$ and $\exists x : s.\phi$ are also formulas.

Semantics

A Σ-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 $I(f) : I(s_1) \times I(s_2) \times \ldots \times I(s_k) \to I(s_{k+1})$. A variable assignment β under a Σ-structure m maps each variable $x \in X$ to an element $\beta(x) \in I(\alpha(x))$.

The recursive evaluation function $val_{m,\beta}(\cdot)$ is defined for terms and formulas. We omit the cases for logical constants and basic logic connectives, because they are identical to the cases in 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 $i \in \{1, 2, \ldots, n\}$
- $val_{m,\beta}(\exists x : s.\phi) = 1$ if there exists an assignment β' differing from β at most in the value of x of sort s, such that $val_{m,\beta'}(\phi) = 1$. Otherwise, $val_{m,\beta}(\exists x : s.\phi) = 0$.
- $val_{m,\beta}(\forall x : s.\phi) = 1$ if no assignment assignment β' exists, such that β' differs from β at most in the value of x of sort s and $val_{m,\beta'}(\phi) = 0$. Otherwise, $val_{m,\beta}(\forall x : s.\phi) = 1$.

A theory T is a pair (Σ, M) of a multi-sorted signature Σ and a class of Σ-structures M. For example, real arithmetic and floating-point arithmetic are theories in this sense. A formula ϕ is T-satisfiable if there is a structure $m \in M$ and a variable assignment β such that ϕ evaluates to 1; we denote this by $m, \beta \models_T \phi$, and call (m, β) a T-solution of ϕ. If formula ϕ is said to be *valid* if it evaluates to 1 in every Σ-structure m and for every assignment β.

5. The Satisfiability Problem

The problem of determining whether there exists a satisfying assignment for a propositional formula is called the *satisfiability* or 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 Boolean constraints can be satisfied. The SAT problem is computationally hard, in the sense that an efficient (i.e. a deterministic polynomial complexity) algorithm is unknown. Moreover, the SAT problem is the first problem to be proven NP-complete, by Cook [22]. Levin independently discovered the same result [50], and the result is nowadays known as the Cook-Levin theorem.

In recent years, a 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. The first procedure for solving the SAT problem named DP [27] after its authors David and Putnam. The DP procedure was later extended into what is known as the DPLL procedure [26], named after its authors Davis, Putnam, Logeman and Loveland. Its modern version is called the *conflict driven clause learning* algorithm, and it leverages some very deep insights and highly efficient data structures to improve upon the original algorithm. 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 conjunctive normal form (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 conjunction of clauses to be satisfied, represented as a set. Clauses are disjunctions of literals and due to CNF structure, at least one literal needs to be assigned value *true* to satisfy the formula. An empty

set of clauses is considered satisfiable. A formula containing an empty clause, denoted $()$, 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). A literal l is called *pure* if $\neg l$ 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 Davis-Putnam Procedure

The Davis-Putnam (DP) procedure for propositional satisfiability is named after the authors Martin Davis and Hillary Putnam [27]. 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 the Algorithm 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: $(A \vee p) \wedge (B \vee \neg p) \wedge R$, where A, B and R contain no occurrences of p and $\neg p$. A crucial observation is that formula F' is satisfiable if and only if the formula $(A \vee B) \wedge R$ is satisfiable. The procedure then proceeds to solve formula $(A \vee B) \wedge R$ instead of F'. Removing the variable in this way creates a smaller problem (in number of variables), however, variable elimination results in a quadratic blow-up in each iteration.

5.1.2 The DPLL Procedure

The Davis-Putnam-Logeman-Loveland (DPLL) procedure [26] is built upon the DP procedure but introduces the *split rule*. Instead of eliminating a variable

```
Input  : Formula F in CNF
Output: SAT if F is satisfiable, UNSAT otherwise
begin
    if F is empty then
        return SAT
    end
    if () ∈ F then
        return UNSAT
    end
    if (l) ∈ F then
        return DP (F[l ↦ ⊤])
    end
    if l is pure in F then
        return DP (F[l ↦ ⊤])
    end
    return DP (eliminate_variable (F))
end
```

Algorithm 2: The DP procedure

from the formula, a case analysis is performed. Consider again the formula $F' = (A \vee p) \wedge (B \vee \neg p) \wedge R$ (which is equivalent to F). F' is unsatisfiable if and only if both $F'[p \mapsto \top] = B \wedge R$ and $F'[\neg p \mapsto \top] = A \wedge R$ are unsatisfiable. The procedure first attempts to solve the problem with literal p set to true, i.e., $F[p \mapsto \top]$. If that fails, then it solves F with literal p set to false, i.e., $F[\neg p \mapsto \top]$. The split rule allows dynamic simplification of the formula, unlike the variable elimination step which causes a blow-up in the number of clauses. By making the variable assignments explicit, the search space gains a tree-like structure. The DPLL procedure is shown in the Algorithm 3.

Abstract Transition System for the DPLL algorithm

The performance of the 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 the perspective of the core algorithm. To abstract away such details, the DPLL algorithm can be presented as a transition system [59]. The rules applied in Algorithm 3 are presented as transitions of the system. We distinguish between the search states $\langle M, C \rangle$, and two accepting states *unsat* and *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 deci-

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` $(F[l \mapsto \top])$
  end
  if l is pure in F **then**
    return `DPLL` $(F[l \mapsto \top])$
  end
  Choose a variable p in F
  if `DPLL` $(F[p \mapsto \top]) = SAT$ **then**
    return SAT
  else
    return `DPLL` $(F[\neg p \mapsto \top])$
  end
end

Algorithm 3: The DPLL procedure

sions, since we know the reason why the implied literal needs to be true. The *value* function is used to evaluate a literal under the trail:

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

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

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

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. *unsat* if the C_0 is unsatisfiable, or 2. *sat* if the C_0 is satisfiable

The *backtrack* 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.

Propagate

$$\langle M, C\rangle \longrightarrow \langle [M, l], C\rangle \quad \textbf{if} \quad \begin{array}{l} c = (l_1 \vee \ldots \vee l_m \vee l) \in C \\ \forall i : \mathit{value}(l_i, M) = \mathit{false} \\ \mathit{value}(l, M) = \mathit{undef} \end{array}$$

Decide

$$\langle M, C\rangle \longrightarrow \langle [M, \bullet l], C\rangle \quad \textbf{if} \quad \mathit{value}(l, M) = \mathit{undef}$$

Sat

$$\langle M, C\rangle \longrightarrow \mathit{sat} \quad \textbf{if} \quad \begin{array}{l} M \text{ is complete} \\ \mathit{value}(c, M) = \mathit{true} \text{ for all } c \in C \end{array}$$

Unsat

$$\langle M, C\rangle \longrightarrow \mathit{unsat} \quad \textbf{if} \quad \begin{array}{l} M \text{ contains no decisions} \\ \mathit{value}(c, M) = \mathit{false} \text{ for some } c \in C \end{array}$$

Backtrack

$$\langle [M, \bullet l, N], C\rangle \longrightarrow \langle M, \neg l, C\rangle \quad \textbf{if} \quad \begin{array}{l} N \text{ contains no decisions} \\ \mathit{value}(c, M) = \mathit{false} \text{ for some } c \in C \end{array}$$

Figure 5.1. The rules of the DPLL transition system

5.1.3 The CDCL Procedure

In the past couple of decades, the implementation and understanding of the DPLL procedure has reached new levels. Between some crucial insights and highly specialized data structures, the algorithm has become even more streamlined and efficient. The conflict-driven clause learning (CDCL) algorithm has two crucial improvements: *conflict analysis* and *learning*. Figure 5.2 shows the transition system of the CDCL algorithm, but before we get to the rules, we discuss improvements over the DPLL algorithm.

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, *Boolean resolution* [62] is applied 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 the trail elements. At some point during the backtracking, the conflict clause either becomes an implication or

there are no more decisions on the trail. 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*. Learning is instrumental to shortcut repeated inference of the same information, such as detecting the conflict involving the same variables over and over again, due to backtracking taking place in the meantime. Over time learned clauses can become detrimental to performance due to upkeep overhead, so some of them can be dropped, i.e., we *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 addition to the DPLL algorithm are the *restarts*. During the search it is possible to get stuck in some part of the search space. By restarting the search while retaining the learned clauses, we can avoid the problematic search space and discover a solution elsewhere. The *restart* rule allows the search to continue from an empty partial model. This rule is dangerous in the sense that it affects termination and completeness of the algorithm. To avoid those problems, a number of restart schemes have been devised. For example, the Luby restart scheme is a non-uniform restart strategy, that is obtained by concatenation of prefixes of a geometric series in a particular way [43].

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*. Again, a number of heuristics plays a crucial role in the performance of the algorithm [7], such as the choice of variable to decide upon (e.g., VSIDS heuristic[56]), which clauses to learn and which to forget, how often should the search be restarted.

The CDCL transition system

The CDCL transition system is fairly similar to that of the DPLL algorithm. The procedure consists of two phases, search and conflict analysis. The set of states is expanded to reflect this and consists of:

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

In order to support conflict analysis, the trail distinguishes two kind 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.

Propagate

$$\langle M, C\rangle \longrightarrow \langle [M, c \to l], C\rangle \quad \textbf{if} \quad \begin{array}{l} c = (l_1 \vee \ldots \vee l_m \vee l) \in C \\ \forall i : value(l_i, M) = false \\ value(l, M) = undef \end{array}$$

Decide

$$\langle M, C\rangle \longrightarrow \langle [M, \bullet l], C\rangle \quad \textbf{if} \quad l \in \boldsymbol{B}, value(l, M) = undef$$

Conflict

$$\langle M, C\rangle \longrightarrow \langle M, C\rangle \vdash c \quad \textbf{if} \quad c \in C, value(c, M) = false$$

Sat

$$\langle M, C\rangle \longrightarrow sat \quad \textbf{if} \quad \begin{array}{l} M \text{ is complete} \\ \forall c \in C : value(c, M) = true \end{array}$$

Unsat

$$\langle M, C\rangle \longrightarrow unsat \quad \textbf{if} \quad \begin{array}{l} M \text{ contains no decisions} \\ \exists c \in C : value(c, M) = false \end{array}$$

Resolve

$$\langle [M, d \to l], C\rangle \vdash c \longrightarrow \langle M, C\rangle \vdash r \quad \textbf{if} \quad \begin{array}{l} \neg l \in c \\ r = resolve(c, d, l) \end{array}$$

Consume

$$\langle [M, d \to l], C\rangle \vdash c \longrightarrow \langle M, C\rangle \vdash c \quad \textbf{if} \ \neg l \notin c$$

$$\langle [M, l], C\rangle \vdash c \longrightarrow \langle M, C\rangle \vdash c \quad \textbf{if} \ \neg l \notin c$$

Backjump

$$\langle [M, N], C\rangle \vdash c \longrightarrow \langle [M, c \to l], C\rangle \quad \textbf{if} \quad \begin{array}{l} c = l_1 \vee \ldots \vee l_m \vee l \\ \forall i : value(L_i, M) = false \\ value(l, M) = undef \\ N \text{ starts with a decision} \end{array}$$

Learn

$$\langle M, C\rangle \vdash c \longrightarrow \langle M, C \cup \{c\}\rangle \vdash c \quad \textbf{if} \ c \notin C$$

Forget

$$\langle M, C\rangle \longrightarrow \langle M, C \setminus \{c\}\rangle \quad \textbf{if} \ c \in C \text{ is a learned clause}$$

Restart

$$\langle M, C\rangle \longrightarrow \langle [], C\rangle \quad \textbf{if}$$

Figure 5.2. The rules of the CDCL transition system

The rules of the CDCL transition system are shown in Fig. 5.2. The main difference to the DPLL transition system is that the rules are 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 (*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, some are can be encoded more naturally in different theories. The *satisfiability modulo theories* (SMT) is the problem of determining satisfiability of a formula in the presence of background theories. The background theories depend 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, etc. [31]. The advantage of modeling a problem using SMT rather than SAT is usually in expressiveness, ease of modeling and scalability (many encodings into propositional logic have a super-linear spatial complexity).

There are several approaches to the SMT problem. A straightforward way is called *eager SMT*, and entails translation of the problem into an equisatisfiable propositional formula, and using a SAT solver [48]. 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 [49] 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 lazy approach keeps the Boolean and the theory world separated, which allows the solver to be engineered in a modular way. Communication between the solvers is through a simple interface. The theory solver must be able to determine the satisfiability of a conjunction of theory literals. An advantage of a dedicated theory solver is that one can use tailor-made procedures suitable for the problem at hand, rather then relying on the ability of the SAT solvers to handle the encoding. It is possible to have multiple theory solvers which cooperate in the case of theory combinations. In the DPLL(T) architecture, this is often achieved using Nelson-Oppen theory combination [58]. To solve constraints over several theories, the constraints are rewritten and partitioned into constraints of individual theories. Decision procedures work over constraints of their respective theories and only share constants, which are used to propagate knowledge between the theories in the form of equalities.

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) [59].

T-Propagate

$$\langle M, C\rangle \longrightarrow \langle [M, e \to l], C\rangle \quad \textbf{if} \quad \begin{array}{l} value(l, M) = undef \\ l_1, l_2, \ldots, l_n \in M \\ l_1, l_2, \ldots, l_n \models_T l \\ e = (\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n \vee l) \end{array}$$

T-Conflict

$$\langle M, C\rangle \longrightarrow \langle M, C\rangle \vdash c \quad \textbf{if} \quad \begin{array}{l} l_1, l_2, \ldots, l_n \in M \\ l_1, l_2, \ldots, l_n \models_T \bot \\ c = (\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n) \end{array}$$

Figure 5.3. Theory rules of DPLL(T)

5.2.1 Beyond DPLL(T)

Despite many advances, the DPLL(T) approach has several limitations that newly emerging approaches try to alleviate. We consider two lines of work generalizing the DPLL(T) framework. The *abstract CDCL* generalizes the CDCL algorithm to work with lattices of variable assignments and conflict explanations. The other approach brings together the DPLL(T) framework and the model constructing approaches in a framework, called mcSAT, that generalizes both. These new approaches bring to light interesting new aspects of the SMT problem and the DPLL(T) framework.

ACDCL — abstract conflict driven clause learning

The ACDCL algorithm [34] generalizes the CDCL algorithm, by extending its primitives to work over abstract domains. In certain abstract domains meet operations can model conjunction precisely, while join operations over-approximate disjunction. The partial model obtained during the CDCL search is a conjunction of facts (currently assumed or inferred to be true). Constraint propagation can therefore be seen as fixpoint iteration over an abstract domain. On the other hand, conflict analysis synthesizes a conflict clause, i.e. a disjunction. The conflict analysis can be seen as an under-approximate transformer

over an abstract domain of conflict clauses. This view opens new ways to reason about concrete domains, by implementing these operations over abstract domains instead. The ACDCL algorithm has been instantiated for the theory of FPA using the interval abstract domain [11] and shown great improvements in performance compared to the usual DPLL(T) approach.

mcSAT — model constructing satisfiability calculus

The DPLL(T) framework separates the knowledge between Boolean and theory worlds. Essentially, the SAT solver conducts the high-level search and consults theory solvers as needed. Theory learning is often limited by the facts present in the original problem, which means that the learning takes place on the Boolean level, i.e. in the SAT solver. *Model-constructing* approaches are an alternative approach to the DPLL(T) framework, and are also called *natural domain* SMT [23, 47, 53]. Model constructing approaches operate directly on models of the background theory, and modify them until they satisfy the constraints. The model constructing approaches are very efficient in their respective theories, but cannot compare with the DPLL(T) when it comes to Boolean reasoning.

The model-constructing satisfiability calculus (mcSAT) [32] is a framework that can express both DPLL(T) and the model constructing approaches. It 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. The trail represents a feasible partial model during the search. If the trail becomes infeasible, i.e. it cannot be extended to a model of the constraints, conflict analysis should begin. The partial model enables better theory propagation, since the variable assignments are visible to both the Boolean and theory solver. To facilitate detection of infeasability of the partial model, all the theory literals on the trail should be satisfied by model assignments before new Boolean decisions are made. If that is the case, then there are no conflicts between the partial model and asserted constraints and the search can proceed. Otherwise, a conflict is detected and conflict analysis can begin. The mcSAT framework allows conflict analysis to return true theory lemmas as explanations of a conflict. These lemmas can feature newly introduced literals, which is not common in DPLL(T) implementations. mcSAT-style approach has been applied to linear integer arithmetic [46], nonlinear real arithmetic [45]. In this thesis we present an instantiation of mcSAT for the theory of quantifier-free bit-vectors [64].

The mcSAT framework relaxed separation between the Boolean and theory worlds, but did not provide mechanisms for general theory combination. Recent work by Bonacina et al., generalizes mcSAT further to a framework that allows Nelson-Oppen style theory combination, called *conflict-driven satisfiability calculus* (CDSAT) [10].

6. Reasoning about Machine Arithmetic

The work presented in this thesis builds on research into: decision procedures for bit-vector arithmetic and floating-point arithmetic, approximations and abstractions for reasoning about machine arithmetic and work on natural domain reasoning. We start with an overview of existing decision procedures, consider the paradigm of natural-domain reasoning and discuss more recent developments.

6.1 Decision Procedures for Bit-Vector Arithmetic

The theory of quantifier-free bit-vector arithmetic is used to describe the behavior and operations of bounded integer arithmetic. Bit-vectors are parametric in their length, meaning that there are infinitely many bit-vector sorts in the theory. The core theory of bit-vectors consists of constant literals 0_n and 1_n, operations *concatenation* $(\cdot)$ and *extraction* $(\mathsf{extract}_p^n)$ and *equality* predicate $(=)$. In addition to the core theory, the full theory consists of bit-wise negation, arithmetic and logical shifts, signed and unsigned comparison, addition, subtraction, multiplication, division, bit-wise conjunction, bit-wise disjunction, bit-wise exclusive disjunction, etc. These operations can be grouped into: 1. string operations (concatenation and extraction), 2. logical operations (bitwise operations), and 3. arithmetic operations (addition, subtraction, etc).

The state-of-the-art in bit-vector solving first uses *pre-processing* techniques to simplify the problem by producing a simpler equisatisfiable formula. When applied during the solving, these techniques are called *in-processing*. For example, MathSAT employs more than 300 pre-processing rules. For a detailed description of these techniques we refer you to Franzén's dissertation [36]. After the formula has been pre-processed, one of the following approaches is applied.

The Eager Approach

The encoding of bit-vector arithmetic into the propositional logic is called *flattening* or *bit-blasting* [48]. For each bit-vector variable v of size n, n new Boolean variables $v_0, v_1, \ldots, v_{n-1}$ are introduced. Atomic constraints of bit-vector arithmetic are expressed in terms of their corresponding Boolean variables. Bit-blasting is typically done upfront, i.e., eagerly. However, this approach scales rather poorly. Namely, encoding of some operations, such as multiplication, is quadratic in size of the bit-vector, which quickly leads to

formulas that are too big for the solver to handle. Eager approaches, in general, cannot determine what is relevant for the problem and have to encode everything. This especially becomes an issue when the propositional formula becomes too large to be stored in memory. Furthermore, by bit-blasting word-level information is lost. The SAT solver is unaware of relationship between individual Boolean variables, e.g., whether a variable represents a parity or a sign bit. While the loss of information hampers reasoning about arithmetic operations, the propositional encoding leverages fully the advances in SAT-solving technology. Most solvers for the quantifier-free theory of bit-vectors implement eager bit-blasting, Z3 [30], Yices [35], MathSAT [18], Boolector [13], CVC4 [4], etc.

The Lazy Approach
Instead of encoding all terms upfront, the lazy approach delays bit-blasting of terms until they become relevant for the search. The lazy approach to bit-vectors was pioneered by Bruttomesso et al [14] and was first implemented in MathSAT. Hadarean et al. have successfully combined the eager and lazy approaches into a portfolio [40]. The two paradigms are complementary and together improve performance, both in terms of solving time and number of solved instances. The lazy solver is integrated into the DPLL(T) framework Nelson-Oppen style, and targets the following fragments: core theory of bit-vectors, bit-vector inequalities, and equational fragments. An approximation refinement approach for bit-vector arithmetic is implemented in the UCLID system [16]. Unlike bit-blasting, these approaches are compatible with theory combination frameworks. Various solvers implement specialized sub-solvers, usually for equality, e.g. Z3 and Yices [30, 35].

Word-Level Approaches
To mitigate the information loss due to bit-level reasoning, several methods attempt to reason on *word-level*. Compared to bit-blasting, all these approaches qualify as lazy. For instance, the core theory can be solved in polynomial time by reduction to the theory of equality [15]. A fragment of the bit-vector theory free of bit-wise operations can be solved by reasoning about modular arithmetic [2]. An approach lifting stochastic local search to the theory of bit-vectors can outperform bit-blasting [37].

In this thesis, a model-constructing solver for the theory of bit-vectors, named mcBV, is presented. The approach uses intervals and bit-patterns to represent bit-vectors. Recently, a model-constructing approach, complementary to mcBV which is presented in this thesis, has been proposed [39]. As a complement to intervals and bit-patterns, used by mcBV, they propose to represent bit-vectors as Binary Decision Diagrams (BDDs) and describe learning mechanisms over BDDs.

Chihani et al. propose a CDCL-style learning mechanism that operates directly on word-level [17]. Their procedure lifts learning mechanisms from

SAT solving and integrates ideas from propagators used in the constraint programming community.

mcBV

Presented in this thesis is an implementation of the model-constructing calculus for BVA, mcbv. It can be categorized as a lazy word-level solver. It uses two over-approximations of bit-vectors — *bit-patterns* and *intervals*. These representations are word-level, compact and designed to capture Boolean and arithmetic properties, respectively. Bit-patterns are run-length encoded bit-vectors of 3-value bits: zero, one and unknown. The run-length encoding of unknown bits essentially pre-empts bit-blasting. Unknown bits are transformed into ones and zeroes when they become relevant to the current trail state. As a consequence, the procedure does not explicitly bit-blast, but in the worst case the bit-pattern representation is equivalent to bit-blasting. In addition to Boolean learning, mcBV uses two flavors of greedy conflict generalization tuned for arithmetic and Boolean constraints. Overall, mcBV cannot compete with state-of-the-art bit-vector solvers in terms of runtime, however, it does outperform it on several classes of benchmarks.

6.2 Reasoning about Floating-Point Arithmetic

Depending on the context, reasoning about floating-point arithmetic takes different forms. When it comes to verification, the focus is on decision procedures. In other areas, various kinds of analyzers might be preferable. The related work presented here is broadly categorized into decision procedures for FPA and analysis tools for FPA.

6.2.1 Decision Procedures for Floating-Point Arithmetic

Decision procedures for FPA have evolved over the years to rely more and more on approximations. Here we give an overview of reduction of FPA to SAT, ACDCL algorithm for FPA and several approximation-based decision procedures.

Reduction to SAT using bit-blasting

If we consider the representation of FPA in the IEEE-754 standard, it becomes obvious that it can be reduced to BVA. A normal floating-point number v is a triplet of bit-vectors $(v_{sgn}, v_{exp}, v_{sig})$, which represents the value $(-1)^{v_{sgn}} \cdot v_{sig} \cdot 2^{v_{exp}}$. All floating-point operations can be expressed as a combination of operations over the sign, significand and exponent. For example, floating-point multiplication of two numbers becomes multiplication of signs

and significands and addition of exponents, i.e.:

$$\begin{array}{lllll} & (-1)^{a_{sgn}} & \cdot & a_{sig} & \cdot & 2^{a_{exp}} \\ \otimes_{rm} & (-1)^{b_{sgn}} & \cdot & b_{sig} & \cdot & 2^{b_{exp}} \\ \hline round(rm, & (-1)^{a_{sgn}+b_{sgn}} & \cdot & a_{sig} \cdot b_{sig} & \cdot & 2^{a_{exp}+b_{exp}}) \end{array}$$

The above equation conveys the core idea, but it obscures several details, such as the leading implicit bit, normalization of representation, etc. For detailed description of various operations see the Handbook of Floating-Point Arithmetic [57].

The reduction of FPA to BVA is achieved by decomposing all FPA operations into their definitions over the sign, significand and exponent. The resulting bit-vector formula is then bit-blasted and given to a SAT solver.

This approach suffers from all the downsides of bit-blasting, high-level arithmetic information is lost and scalability quickly becomes an issue. It is interesting to note that different operations scale differently with the size of the exponent and the significand. In the multiplication equation, we can see that the resulting exponent is obtained by addition, therefore it is linear in respect to the exponent size, while the significand requires a quadratic encoding due to the multiplication involved. In addition, handling special values, normalization and rounding introduces additional additional Boolean structure in the final formula.

Approximation-Based Approaches

Approximation based approaches are often inspired by approaches in model checking, with CEGAR (Counter-Example Guided Approximation Refinement) [20] being one of the common automated approximation refinement schemes. CEGAR produces a series of abstractions that are refined based on spurious counter-examples. In general, CEGAR-style schemes depend on approximations which preserve solutions and counter-examples, called over- or under-approximations respectively. An over-approximation $\bar{\phi}$ of ϕ is a formula such that any model m of ϕ is also a model of $\bar{\phi}$. Formula $\bar{\phi}$ captures all models of ϕ and possibly others. As a consequence, if the over-approximation $\bar{\phi}$ has no solutions, then the original formula has no solutions either. An under-approximation $\hat{\phi}$ of ϕ is a formula such that any counter-example c of ϕ is also a counter-example of $\hat{\phi}$. Over- and under-approximations have shown themselves very useful in variety of domains and have been termed *abstractions* or *conservative approximations*.

Mixed abstractions

Brillout et al. proposed a CEGAR-style approach for solving FPAformulas that combines over- and under-approximations, called *mixed abstractions* [12]. The approximation reduces the precision of the significand in FPA operations, i.e. the number of significand bits. Operations are over-approximated by

considering results of the operations under any rounding mode. Conversely, under-approximations are obtained by assuming that no rounding takes place. Two schemes using alternating abstractions and truly mixed abstractions have been evaluated, and shown to improve the solving time considerably in respect to a naive bit-blasting approach.

The ACDCL algorithm for FPA
The ACDCL algorithm has been successfully applied to FPA [11]. The ACDCL algorithm works over abstract domains. In particular, instead of reasoning in the domain of floating-point numbers, it reasons over the domain of intervals. The algorithm uses an incomplete proof procedure, interval constraint propagation (ICP). However, since the floating-point domain is finite, complementing ICP with case splitting yields a sound and complete decision procedure. The algorithm splits the intervals in turn, allowing propagation to take place and learning from conflicts that arise.

Proxy theories
Ramachandran et al. [60] present an approximation-refinement loop very similar to the work presented in this thesis. Their framework reduces the input problem into a problem of a simpler *proxy theory*. They present several approximations of FPA using real arithmetic. The solution of a proxy theory problem is used as seed to construct a model of the original problem. The main contribution is in an aggressive reconstruction step that, whenever the constructed model violates the constraints, identifies simple univariate constraints which are used to correct the model. This technique is called *numeric model lifting*. They consider several proxy theories, pure real arithmetic, mixed real and floating-point arithmetic and reduced-precison FPA.

One of the biggest challenges for approximation using reals, is the lack of representation for the special values. Initially, the approximation consists of real constraints entirely, and if refinement is necessary, constraints that fail to be satisfied after numeric model lifting are not approximated in future iterations. In addition, they use δ*-complete procedures* for approximate model search, which are useful on problems that are satisfiable over FPAbut unsatisfable over reals.

Systematic approximation refinement in UppSAT
The systematic approximation refinement framework presented in this thesis draws heavy inspiration from the work on mixed abstractions. The traditional requirement for abstractions is abandoned, since creating conservative approximations for FPA is fairly difficult. This departure allows approximations that reduce the size of both the exponent and the significand, which reduces the propositional encoding even further than that of mixed abstractions. Despite the use of non-conservative approximations, the systematic approximation refinement framework is general and preserves soundness, completeness and

termination of the underlying decision procedure. Furthermore, by extracting unsatisfiable cores of unsatisfiable approximations, the framework mitigates the lack of conservative properties of approximations in the case of unsatisfiable problems.

UppSAT, a new implementation of the framework, offers an environment for modular design of approximations and easy integration with SMT solvers as the underlying decision procedure. In UppSAT, approximations are composed of modular components that can be combined with minimal modifications. The encoding of formulas and decoding of values, depends on the nature of the approximation, i.e. whether we are approximating FPA with reals or vice versa. The strategies are more general, e.g., only tangentially depend on the theories involved. UppSAT defines templates for some relatively simple, but in practice quite effective strategies, and enables the user to combine them easily. For instance, by using the templates, specification of reduced precision floating-point approximation needs less than 300 lines of Scala code. UppSAT has been used to approximate FPA using reduced-precision FPA, reals and fixed-point arithmetic (encoded as BVA). As back-end decision procedures we used bit-blasting and non-linear real solver (nlsat) provided by Z3 and bit-blasting and ACDCL provided by MathSAT.

6.2.2 Other Approaches to Reasoning about FPA

Besides decision procedures, there is a rich body of research in the form of tools and analyzers that focus on numeric properties of executions and highlight potential problems for the user. These tools require a fair amount of expertise to use.

In many safety-critical applications, correct estimate of round-off errors is essential. Fluctuat [38] is based on abstract domains and integrates widening techniques to handle iterative programs. Astrée [24] is a static analyzer that uses abstract domains to detect run-time exceptions in floating-point programs. RangeLab [52] gives bounds on the round-off error for basic arithmetic operations over floats using intervals. FPTaylor [63] approximates floating-point expressions using Taylor expansions to obtain error bounds and provides certificates for HOL Light [41]. VCFloat [61] is another tool that uses interval arithmetic to compute round-off errors with certificates in Coq [33]. Gappa [28, 9] is a semi-automatic approach to compute bounds using interval arithmetic and it produces proof certificates in Coq. Rosa [3] is a tool that computes the sufficient precision of floating-point numbers necessary so that round-off errors are within specified margin. It starts with reals, i.e. infinite precision floating-point numbers and reduces the precision as long as the round-off error stays within given bounds. Interval-based methods are quite efficient in terms of runtime, but can yield rather pessimistic bounds, unless they use additional techniques such as optimization.

7. Contributions

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

An alternative to the DPLL(T)-style SMT solvers are model constructing procedures. The model constructing satisfiability calculus (mcSAT)[32] 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 I 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.

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 the 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 had limited success when applied to the floating-point arithmetic. Approximations which are not over- nor under-approximations can find incorrect solutions or overlook correct solutions. Paper II 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.

UppSAT, a new implementation of the approximation refinement framework is presented in Paper III. UppSAT is an abstract approximating SMT solver, which can be instantiated with an approximation and a back-end SMT solver to yield an approximating SMT solver. Approximations in UppSAT have a modular design, which provides a perfect environment for developing and testing new approximations. Implementation of approximations of FPA using reduced precision floating-point arithmetic, fixed-point arithmetic and real arithmetic in UppSAT is presented. An evaluation of these approximations in combination with several back-end SMT solvers is presented and results show significant speed-up for the bit-blasting back-ends.

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 and approximations for the particular domain has proven crucial for the procedure. Finding those approximations is far from easy. However, if their properties are exploited, results are significant improvements in performance. To make experimentation with approximations easier, the approximation refinement framework is implemented as an abstract SMT solver. UppSAT is designed to enable easy swapping of the underlying decision procedure, which allows the user to easily evaluate how a particular approximation 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.

9. Sammanfattning på Svenska

Datorsystem genomsyrar det moderna samhället: transport, kommunikation, ekonomi, stat och hushåll förlitar sig på dussintals av dem. I den bästa av världar så fungerar dessa system som tänkt, men i verkligheten kommer några av dem inte att göra det. I många fall är konsekvenserna av fel inte så allvarliga. Att vara utestängd från e-post eller sociala medier en stund är trots allt bara irriterande. Vissa system har däremot en kritisk roll i styrningen av ett större system, exempelvis för navigation eller för att garantera säkerhet. Fel i sådana system kan vara förödande, de kan kosta tid och pengar eller till och med riskera liv.

Betydande resurser läggs på att förhindra att fel kan inträffa, men det är mycket svårt att säkerställa att datorsystem fungerar korrekt. Datorsystem består av både hårdvara och mjukvara som arbetar ihop, och dessa måste fungera korrekt, både oberoende av varandra och tillsammans. Det finns många felkällor: ett fel i hårdvarudesignen låg bakom Pentium FDIV-buggen, en brist på insikt om hur avrundning av tal fungerar orsakade misslyckandet med Patriotmissilen, medan olyckan med Ariane 5 berodde på ett misstag i den övergripande systemdesignen. Istället för att garantera att ett helt system fungerar korrekt, vilket ofta är ogenomförbart på grund av dess storlek ock komplexitet, så läggs fokus på delsystem med säkerhetskritiska roller.

Formella metoder används mer och mer för specifikation, implementation och kvalitetskontroll av datorsystem. Formella metoder är en grupp av tekniker och metoder som är baserade på formella matematiska modeller. De kan användas för att bli av med oklarheter i specifikationer, för att syntetisera korrekta implementationer direkt från specifikationer eller för att bevisa att en implementation uppfyller vissa säkerhetskriterier. För att kunna bevisa att en implementation uppfyller sådana kriterier måste vi kunna resonera kring datoraritmetik. Detta formuleras i regel som ett SMT-problem (Satisfiability Modulo Theories), där de relevanta teorierna för datoraritmetik är de om bitvektorer och flyttalsaritmetik. De metoder som finns för att resonera kring sådana teorier lider av dålig skalbarhet. De kan därför fungera bra på små problem, men kan inte hantera problem som är större.

Den här avhandlingen utforskar approximationer som ett sätt att utöka beslutsmetoder, detta genom ett generellt ramverk för att förfina approximationer. Ramverket löser en serie av approximationer, vilka är förenklade versioner av det ursprungliga problemet. Dessa är till en början väldigt grova, så approximationerna är lätta att lösa. Resultaten från en sådan lösning används sedan för att antingen rekonstruera en lösning till ursprungsproblemet, för att

bevisa att ingen lösning finns, eller för att förfina approximationen. Ramverket bevarar viktiga egenskaper hos den underliggande beslutsmetoden, som sundhet, fullständighet och termination. Sundhet garanterar att bevisen är korrekta. Fullständighet garanterar att allt som är sant inom teorin också kan bevisas. Termination garanterar att till sist kommer antingen en lösning att hittas, eller ett bevis för att ingen lösning finns. Uppsats II beskriver ramverket i detalj. Uppsats III ger en detaljerad utvärdering av ramverkets inverkan på de bästa tillgängliga beslutsmetoderna inom SMT för flyttalsaritmetik.

I den här avhandlingen utforskas även en ny metod som kallas mcBV för att resonera kring teorier om bitvektorer med fast bredd. Den är en variant av mcSAT som använder en ny lat representation av bitvektorer som gör det möjligt att resonera både på bit- och ord-nivå. Den använder en girig mekanism som är kapabel till mer generellt lärande än andra existerande metoder. En utvärdering av mcBV visar att den kan prestera bättre än bit-blasting på flera klasser av problem. Uppsats I beskriver metoden i detalj och presenterar de experimentella resultaten.

References

[1] Robert A. 1940-(Robert Alexander) Adams. *Calculus: a complete course*. Pearson/Addison Wesley, Toronto, Ont, 6. edition, 2006.

[2] Domagoj Babić and Madanlal Musuvathi. Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005.

[3] Marek Baranowski, Ian Briggs, W Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Moving the needle on rigorous floating-point precision tuning. In *6th Workshop on Automated Formal Methods (AFM 2017)*, 2017.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. CAV*, volume 6806 of *LNCS*. Springer, 2011.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard version 2.6. 2010.

[6] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. ios press, 2009.

[8] Michael Blair, Sally Obenski, and Paula Bridickas. Patriot missile defense: Software problem led to system failure at dhahran. Technical report, Saudi Arabia. Technical Report GAO/IMTEC-92-26, United States Department of Defense, General Accounting office, 1992.

[9] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *Intelligent Computer Mathematics (Calculemus); MKM/CICM*, volume 5625 of *LNCS*. Springer, 2009.

[10] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Satisfiability modulo theories and assignments. In *Proceedings of the Twenty-Sixth Conference on Automated Deduction (CADE)(Lecture Notes in Artificial Intelligence), Leonardo de Moura (Ed.), Vol. to appear. Springer, Berlin, Germany, EU*, 2017.

[11] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *FMSD*, 2013.

[12] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*. IEEE, 2009.

[13] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *LNCS*. Springer, 2009.

[14] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered smt (bv) solver for hard industrial verification problems. In *CAV*, volume 4590, pages 547–560. Springer, 2007.
[15] Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *ICCAD*. ACM, 2009.
[16] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, volume 4424 of *LNCS*. Springer, 2007.
[17] Zakaria Chihani, François Bobot, and Sébastien Bardin. Cdcl-inspired word-level learning for bit-vector constraint solving. 2017.
[18] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
[19] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.
[20] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*. Springer, 2000.
[21] Tim Coe. Inside the pentium-fdiv bug. *DR DOBBS JOURNAL*, 20(4):129, 1995.
[22] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.
[23] Scott Cotton. Natural domain SMT: A preliminary assessment. In *FORMATS*, pages 77–91. Springer, 2010.
[24] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *ESOP*, volume 3444 of *LNCS*. Springer, 2005.
[25] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
[26] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
[27] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
[28] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.
[29] Stijn de Gouw, Jurriaan Rot, Frank S de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java. utils. collection. sort () is broken: the good, the bad and the worst case. In *International Conference on Computer Aided Verification*, pages 273–289. Springer, 2015.
[30] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*. Springer, 2008.
[31] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: An

appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36. Springer, 2009.
[32] Leonardo de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *VMCAI*, 2013.
[33] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
[34] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *POPL*. ACM, 2013.
[35] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *LNCS*, pages 737–744. Springer, July 2014.
[36] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, Italy, 2010.
[37] Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *AAAI*, pages 1136–1143, 2015.
[38] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *SAS*, volume 6, pages 18–34. Springer, 2006.
[39] Stéphane Graham-Lengrand and Dejan Jovanović. An mcsat treatment of bit-vectors (preliminary report). In *SMT 2017-15th International Workshop on Satisfiability Modulo Theories*, 2017.
[40] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *CAV*, volume 8559 of *LNCS*. Springer, 2014.
[41] John Harrison. The HOL light manual (1.1), 2000.
[42] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
[43] Jinbo Huang et al. The effect of restarts on the efficiency of clause learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.
[44] IEEE Comp. Soc. *IEEE Standard for Floating-Point Arithmetic 754-2008*, 2008.
[45] Dejan Jovanović and Leonardo De Moura. Solving non-linear arithmetic. *ACM Communications in Computer Algebra*, 46(3/4):104–105, 2013.
[46] Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1), 2013.
[47] Konstantin Korovin, Nestan Tsiskaridze, and Andrei Voronkov. Conflict resolution. *Principles and Practice of Constraint Programming-CP 2009*, pages 509–523, 2009.
[48] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view*. Springer Science & Business Media, 2008.
[49] Shuvendu K Lahiri and Sanjit A Seshia. The uclid decision procedure. In *International Conference on Computer Aided Verification*, pages 475–478. Springer, 2004.
[50] Leonid A Levin. Universal sequential search problems. *Problemy Peredachi*

Informatsii, 9(3):115–116, 1973.
[51] Jacques-Louis Lions et al. Ariane 5 flight 501 failure, 1996.
[52] Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-order and symbolic computation*, 19(1):7–30, 2006.
[53] Kenneth McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In *Computer Aided Verification*, pages 462–476. Springer, 2009.
[54] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Conf. on Real Numbers and Computers*, volume 216 of *Information & Computation*. Elsevier, 2012.
[55] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *International Conference on Principles and Practice of Constraint Programming*, pages 524–538. Springer, 2001.
[56] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.
[57] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of floating-point arithmetic*. Springer Science & Business Media, 2009.
[58] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.
[59] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. ACM*, 53(6), 2006.
[60] Jaideep Ramachandran and Thomas Wahl. Integrating proxy theories and numeric model lifting for floating-point arithmetic. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 153–160, 2016.
[61] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 15–26. ACM, 2016.
[62] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science B.V., 2001.
[63] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *International Symposium on Formal Methods*, pages 532–550. Springer, 2015.
[64] Aleksandar Zeljić, Christoph M Wintersteiger, and Philipp Rümmer. Deciding bit-vector formulas with mcsat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 249–266. Springer, 2016.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1603

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-334565

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017