hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

BITVECTORS IN SMT-RAT AND THEIR APPLICATION TO INTEGER ARITHMETICS

Andreas Krüger

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

Additional Advisor:
M.Sc. Gereon Kremer

Aachen, 08.10.2015

Abstract

One of the most prominent approaches in the field of software verification is based on the problem of Satisfiability Modulo Theories (SMT). Deciding an SMT instance means deciding whether a given first-order formula is satisfiable in the context of a fixed background theory. A theory that is specifically designed for software and hardware verification purposes is the theory of fixed-size bitvector logic (BV). It provides an expressive language to reason about many operations that are implemented in state-of-the-art CPUs.

This thesis presents a novel module for the SMT toolbox SMT-RAT that can decide the satisfiability of quantifier-free formulas in bitvector logic. It makes use of a reduction to the well-known propositional satisfiability problem (SAT). This approach, which is often referred to as flattening or bit-blasting, highly benefits from the successes in the development of efficient SAT solvers throughout the last decades.

While flattening is the predominant method for deciding problems in bitvector logic, it is also possible to transfer this idea to different SMT theories, which are less strongly related to SAT. As the second part of this thesis, an application of bit-blasting to the theory of nonlinear integer arithmetics (NIA) is presented, where the properties of integrality and nonlinearity are typically a burden on classical arithmetic decision procedures. Especially on small domains, an encoding to SAT can be a superior alternative.

Instead of directly encoding to SAT, a technique is demonstrated that reduces NIA problems on a small domain to bitvector arithmetic. We further explain how to apply Interval Constraint Propagation (ICP) for reducing the complexity of the created bitvector formulas and thereby improve the performance of the solver.

An evaluation shows that the newly created bitvector module in SMT-RAT already gives decent performance results. On the NIA theory, the proposed reduction to bitvector arithmetic proves to be highly competitive with common arithmetic decision procedures.

Erklärung

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

Andreas Krüger
Aachen, den 08. Oktober 2015

Danksagung

Ich danke den folgenden Personen für ihre Unterstützung während der Erstellung dieser Arbeit:

- Prof. Dr. Erika Ábrahám für die Möglichkeit, meine Masterarbeit in dem spannenden Gebiet des SMT-Solvings zu verfassen, sowie für die hilfreichen Impulse und wertvollen Ratschläge.
- Prof. Dr. Jürgen Giesl für die Bereitschaft, meine Arbeit als Zweitgutachter zu begleiten.
- Meinem Betreuer Gereon Kremer für die kontinuierliche Unterstützung, zahlreiche fruchtbare Anregungen und den produktiven Austausch.
- Florian Corzilius für viel Hilfsbereitschaft und eine fortwährend sehr angenehme Zusammenarbeit.
- Meiner Familie, insbesondere meiner Frau Julia, meinen Eltern und meiner Schwiegermutter, für den starken Rückhalt in dieser Zeit und während meines gesamten Studiums.

Contents

Chapter 1

Introduction

Throughout the last decades, the impact of technology on mankind has increased to an enormous level. We live in a world where we are used to permanently interact with technical systems, consciously or unconsciously, in almost every aspect of our everyday lives. One of the main drivers for this ongoing development is the performance and availability of computerized systems. The omnipresence of software has caused a trend to hand over more and more tasks to the control of machines, since their actions are perceived as more reliable, more predictable and less error-prone than human behavior. The development of self-driving cars or autonomously acting missiles, for example, is a manifestation of the trust our society puts in technology.

However, this progression also burdens software engineers with a much higher responsibility for their products, a challenge that is usually addressed by an exhaustive testing procedure as an integral part of the software development process. Nonetheless, a characteristic of testing is that it can only prove the presence of errors, not their absence. When it comes to heavily safety-critical applications, it is desirable to obtain stronger statements on the accurate functioning of software. This desire has given rise to the approach of verification, i.e. the application of formal methods to software with the aim to prove its correctness on the basis of a mathematical calculus.

The typical procedure of verification consists of three steps: First, the software to be verified is transformed into a mathematical model. Afterwards, the properties that should be proven are formalized on the basis of this model. Typical properties are the unreachability of certain states that are attributed as bad or harmful, the absence of programming mistakes which can make the system behavior nondeterministic, the correctness of the program output under all possible inputs, or the fulfillment of liveness properties expressing that the system always responds within a certain time frame.

Due to the variety of applications and programming languages, the set of constraints varies within different domains and employed techniques. Since the choice of the mathematical model used for verification purposes also determines the spectrum of statements which can be concluded on its basis, it is a natural consequence that there is not a unique modeling scheme that is suited for all practices. Instead, there is a multitude of logics and theories to choose from, under consideration of their respective strengths and weaknesses.

A very important group of problems is the set of problems which ask for the satisfiability of some first-order logic formula φ in the context of a given theory $\mathcal{T}$. Depending on the concrete application, different theories can be suitable as background theory $\mathcal{T}$. Problems of arithmetic nature, for example, may be described using a theory of arithmetics over integer or real num-

bers, while problems with a more logical than arithmetic character may be based on a theory of uninterpreted functions. The union of all problems following this scheme is referred to as the problem of Satisfiability Modulo Theory (SMT). It can be seen as a generalization of the famous satisfiability problem (SAT), which addresses the question of satisfiability of propositional formulas.

During the last years, SAT and SMT solving have gained a lot of attraction. SAT is typically well-known for its theoretical properties in the field of complexity theory: The SAT problem is NP-complete, which states that it is an NP-hard problem on which all other NP-hard problems can be reduced in polynomial time. It is one of the most important open questions of theoretical computer science whether NP-complete problems like SAT can be solved efficiently, i.e. by a polynomial algorithm on a deterministic Turing machine, but the commonly shared belief is that this is not the case. However, this is a theoretical property concerning the SAT problem in its general form, which does not imply that every SAT problem instance is hard to solve. In fact, a lot of progress has been achieved in building automated solvers that operate efficiently on many instances of SAT arising from practical problems.

The advances in the field of SAT solving have also contributed a lot to the area of SMT solving. State-of-the-art SMT solvers are typically based on an internal SAT solver, equipped with additional solvers for the specific background theory $\mathcal{T}$. As a result of this setup, any performance gains achieved in SAT solving also have a direct effect on the SMT solver performance. In addition, there are also important developments in the field of SMT on its own: In 2003, an international initiative named SMT-LIB [BST10] has been built, which aims at the establishment of common standards in the field of SMT. A general syntax for SMT problem specification has been developed, which is now widely supported by different SMT solvers. Moreover, a number of background theories has been precisely specified in terms of their syntax and semantics.

These common standards allow for a direct comparison of SMT solvers, which is the goal of the SMT-COMP contest [CDW15]: Since 2005, SMT solvers have the chance to participate in this annual competition, in which their performance is evaluated on huge amounts of benchmarks in various categories. The results of the last years make it evident that most solvers are still under active development and that a lot of research is performed in the extension and improvement of SMT solvers.

Starting in 2012, the research group Theory of Hybrid Systems at the RWTH Aachen, lead by Prof. Dr. Erika Ábrahám, initiated the creation of an own SMT solving engine named SMT-RAT (Satisfiability-Modulo-Theories Real Algebra Toolbox) [CKJ+15]. Built on C++, SMT-RAT is provided as free software, designed with the objectives extensibility, modularity and flexibility in mind. At its heart, it consists of several solver modules which can be composed to a user-defined strategy. Depending on the configuration, the generated program is either a full-blown SMT solver or only a theory solver which can be embedded into different SMT solving engines.

Regarding the project name SMT-RAT and its development history, it is obvious that SMT-RAT started with a main focus on theories considering nonlinear real and integer arithmetics. However, it is the ambition of the project to extend its capabilities towards other theories. With the development of modules for linear arithmetics and for uninterpreted sort and function symbols, a shift towards a more universal solver has begun. With the contributions of this thesis, we want to further widen the scope of application of SMT-RAT by creating a designated module for the theory of quantifier-free fixed-size bitvectors (QF_BV in SMT-LIB).

The theory of fixed-size bitvectors originates directly from the software and hardware verification domain. A bitvector is a vector of Boolean values and serves as the model of CPU registers, variables or memory areas in general. With its rich set of predicate and function sym-

bols, the theory has been designed to support the simulation of a CPU, imitating the semantics of many operators which can be found in programming languages. Bit-wise operators like AND or XOR, for example, which are commonly used in software development for embedded systems, are hard to represent using an arithmetic theory. In QF_BV, on the other hand, there is a direct equivalent of these operators, such that the operations can be expressed without any encoding overhead.

In addition to these function symbols for bit-wise computations, the theory also offers function symbols for arithmetic calculations, which are tailored to CPU simulation. The semantics of operators like addition or multiplication differ from the semantics of their equivalents in arithmetic theories in respect of handling very big or very small results: In arithmetic theories, the domain is in general unbounded, allowing for arbitrarily high or low values. This setting differs from the situation in a CPU, where numbers are represented as finite bit-sequences with a maximum length. If a computation result exceeds the limits for its representation, the most significant bits are lost, resulting in what is typically referred to as an overflow. Overflows are a common source of programming errors, which makes it crucial to model them accordingly. In the theory of fixed-size bitvectors, overflow effects are directly modeled by the semantics of its function symbols and require no extra encoding constraints. These properties make bitvector logic a perfect fit for many verification purposes, especially in the field of low-level programming.

In the past, several strategies for solving problems in bitvector logic have been investigated. Though, all state-of-the-art SMT solvers for bitvector logic ultimately use the same solving technique, namely an encoding to SAT. The encoding is performed by replacing each bitvector by a vector of propositional variables and expressing all bitvector constraints as constraints on the propositional level. Afterwards, a regular SAT solver searches for a solution on the generated propositional constraints. Although the encoding of some operators creates a remarkable overhead, the efficiency of modern SAT solving has made this approach feasible. Moreover, there is currently no comprehensive alternative solving strategy that can handle the rich expressivity of bitvector logic without restrictions. For that reason, the theory module for bitvector logic that we developed in the context of this thesis is also based on the flattening approach. We applied several optimizations to obtain a good resulting performance.

As a second big part of our work, we examined the bit-blasting technique from a more universal point of view. Although bit-blasting is typically employed to bitvector logic, the general idea of encoding numeric calculations into SAT can also be adapted to other domains. Especially regarding the original focus of SMT-RAT on nonlinear arithmetic problems, we explored whether these kind of problems can also benefit from the bit-blasting strategy. Our studies are mainly driven by two observations: Firstly, many typical solving strategies applied to linear and nonlinear arithmetics are efficient on the domain of real numbers, but forfeit their good performance when being applied to the domain of integers. The reason is that the solving strategies have no inherent way of being restricted to integer results, which necessitates the usage of additional algorithms like branch-and-bound. Secondly, among the satisfiable arithmetic problem instances, there are many cases in which a solution also exists within relatively small bounds. This gives rise for our development of IntBlast, an SMT-RAT module implementing a SAT-based solving strategy for linear and nonlinear integer arithmetics.

Given an integer arithmetic problem, our IntBlast module restricts the range of all input variables to a set of configurable cardinality by introducing new bounds. Within these bounds, we translate the arithmetic constraints into equisatisfiable constraints in bitvector logic. These constraints are checked for satisfiability using our previously developed module in combination with a SAT solver. If a solution is found in the restricted search space, we know that the original problem is satisfiable, and can provide a suitable model for it. If the restriction does not contain

a solution, we can propagate this information to further solving modules, which then perform a search on the remaining search space by classical approaches. We believe that this composition has the potential to combine the benefits of SAT-based solving and arithmetic-based solving to obtain an improvement of the overall performance.

This thesis is organized as follows: Chapter 2 gives a brief introduction into the backgrounds of SMT solving, in particular within SMT-RAT. We also compare the different strategies that have been established for solving bitvector and integer arithmetic problems in Sections 2.3 and 2.4, respectively. Chapter 3 is devoted to the first half of our work, namely the development of an SMT-RAT module for solving problems in bitvector logic. After a short structural overview in Section 3.1, we use Section 3.2 to describe the process of encoding bitvector logic into propositional logic in more detail. We conclude the chapter by illustrating a number of optimizations in our implementation. On this basis, we present our new approach of bitvector-supported solving of integer arithmetic problems in Chapter 4. While Section 4.1 gives a basic understanding of how the module works and how it is composed internally, we use the following Sections to elaborate on several implementation aspects in more detail. Chapter 5 contains the results of several benchmarks to evaluate the performance of our approach. On this basis, we discuss the observed figures and comment on several aspects. We conclude our thesis in Chapter 6 by giving a short summary over our achievements and outlining proposals for future research.

Chapter 2

Background

In this chapter we give a short introduction into SMT solving both in general and in the SMT-RAT solver. To this end, we define the notion of SMT problems and outline the DPLL(T) algorithm, which is the most widespread solving approach. We then present some of the important structural components of the SMT-RAT solving engine and explain its concept of a user-defined solving strategy. The two last sections are devoted to the two kinds of SMT logics that are relevant for our thesis: At first, the quantifier-free logic of fixed-size bitvectors is described, followed by an illustration of the quantifier-free logics of linear and nonlinear integer arithmetic. For both cases, we establish basic syntax and semantics, and summarize typical solving procedures.

2.1 SAT and SMT

2.1.1 SAT and the DPLL Algorithm

The Satisfiability Modulo Theory (SMT) problem is a generalization of the well-known satisfiability (SAT) problem. A SAT problem instance consists of a formula φ in propositional logic, i.e. a formula composed of propositional (Boolean-valued) variables and Boolean connectives like $\neg$, $\wedge$, $\vee$ or $\rightarrow$. Solving the SAT instance means deciding whether an interpretation $\mathfrak{I}$ exists, such that φ holds under $\mathfrak{I}$ (also abbreviated $\mathfrak{I} \models \varphi$).

It is an important result from complexity theory that SAT is an NP-complete problem. This attribution summarizes two properties: First, SAT belongs to NP, the class of problems that can be solved in polynomial time by a nondeterministic Turing machine (NTM). The reason is that an NTM can nondeterministically "guess" an interpretation $\mathfrak{I}$ and verify its correctness in linear time. SAT thereby belongs to the class of decidable problems. Second, SAT is NP-hard: By the Cook-Levin theorem [Coo71, Lev73], all other problems in NP can be reduced to SAT on a deterministic Turing machine (DTM) in polynomial time. It is still one of the biggest open questions in theoretical computer science whether NP is distinct from P, its subclass of problems decidable by a DTM in polynomial time. However, it is a commonly shared hypothesis that the two complexity classes are in fact unequal.

Although the NP-completeness of SAT implies that there is probably no efficient (i.e., polynomial-time) algorithm to solve it, the performance of several SAT solvers proves that algorithms with a bad worst-case complexity can still be very performant on the majority of "real-world" problem instances. Today, most relevant SAT solvers are based on the Davis-Putnam-Logemann-

Loveland (DPLL) [DP60, Lov78] scheme, which is explained below. Before it can be applied, the SAT instance needs to be transformed into Conjunctive Normal Form:

Definition 2.1.1 (Conjunctive Normal Form)**.** *A propositional formula φ is in **Conjunctive Normal Form** if it has the form $\varphi = \bigwedge_{i=1,\dots,n} \bigvee_{j=1,\dots,m_i} L_{i,j}$ for some $n, m_i \in \mathbb{N}$ and literals $L_{i,j}$, where a literal is an atom (a) or its negation ($\neg a$).*

*The disjunctions $\bigvee_{j=1,\dots,m_i} L_{i,j}$ are called **clauses** of φ.*

It should be noted that the requirement of φ being in CNF is not a restriction, because every propositional formula φ can be transformed into an equisatisfiable formula φ' in CNF with only a linear growth in size using Tseitin's transformation [Tse83].

```
1:  function DPLL(propositional formula φ in CNF)
2:      if ¬BCP(φ) then                          ▷ Boolean Constraint Propagation
3:          return unsat
4:      end if
5:      while true do
6:          if ¬DECIDE(φ) then                   ▷ Make decision
7:              return sat
8:          end if
9:          while ¬BCP(φ) do
10:             if ¬RESOLVE_CONFLICT() then      ▷ Backtracking
11:                 return unsat
12:             end if
13:         end while
14:     end while
15: end function
```

Algorithm 1: DPLL

As soon as φ is in CNF, the DPLL algorithm can be applied. A simple version of the DPLL algorithm is illustrated in Algorithm 1. The main procedure is a loop which builds an internal partial assignment α iteratively, using a backtracking approach. In this context, a partial assignment is a partial function from the set of propositional variables in φ to the set $\{0, 1\}$. At the beginning, $\alpha(v)$ is undefined for every variable v.

The function BCP stands for *Boolean Constraint Propagation*: Based on the partial assignment α, BCP tries to infer further assignments that must hold if φ should evaluate to 1. The most important rule here is called *Unit Propagation*: If φ contains a clause with only one literal, its variable must be assigned accordingly, such that the literal evaluates to 1. The authors of [Lov78] also suggest the application of the *Pure Literal Rule*: If a propositional variable occurs only positively or only negatively in φ, it can be assigned the value 1 or 0, respectively, without loss of generality. The function BCP adds the necessary implications to α, or returns *false* if the unsatisfiability of φ under α is implied.

After the application of Boolean Constraint Propagation, a decision on further assignments to α is made by the function DECIDE. If the assignment α is already complete, DECIDE returns *false* and the input is satisfiable. Otherwise, it picks a variable v occurring in φ for which $\alpha(v)$ is still undefined and set $\alpha(v)$ to a chosen value. In its most simplistic version, DECIDE may simply pick the first unassigned variable in φ, or choose randomly among the set of unassigned

variables. Employing a good heuristic here is one of the key factors to obtain a good overall performance.

After the choice has been made, BCP is called again. Whenever the current assignment α implies unsatisfiability, the function RESOLVE_CONFLICT essentially analyzes the conflicting set of clauses and backtracks to an appropriate previous decision level of DECIDE. If the whole decision tree has been explored without success, RESOLVE_CONFLICT returns *false* and the input is unsatisfiable.

By this technique, the DPLL algorithm is mainly an application of branch-and-bound to a binary search tree with deduction rules applied in every step. In its basic version, it still leaves room for several optimizations. Concepts like conflict learning have already found their way into plenty modern implementations of DPLL.

2.1.2 SMT and the DPLL(T) Algorithm

For many applications, the expressivity of propositional logic is insufficient. While the framework of Boolean connectives is general enough, the restriction to propositional variables makes it hard to model constraints over an infinite domain like the set of integer numbers. This gave birth to a generalization of SAT named Satisfiability Modulo Theory (SMT). In SMT, constraints are expressed in context of a theory $\mathcal{T}$, consisting of a domain (like $\mathbb{Z}$) alongside with interpretations for some function or predicate symbols (like $+$, $-$, $\cdot$ or $<$). An SMT problem instance is then composed of the theory $\mathcal{T}$ and a first-order formula φ. In φ, variables are no longer Boolean-valued, but range over the domain of $\mathcal{T}$. Again, φ is said to be satisfiable in $\mathcal{T}$ if an interpretation $\mathfrak{I}$ exists that maps the free variables in φ to elements of the domain of $\mathcal{T}$ such that $\mathfrak{I} \models \varphi$ with respect to $\mathcal{T}$. Since each theory $\mathcal{T}$ creates a whole family of SMT problems, we also refer to these problems as SMT($\mathcal{T}$) instances.

First-order logic in general has a much higher expressivity than propositional logic. On the other hand, a higher expressivity of a logic is often accompanied with worse meta-logical properties: Other than in propositional logic, the general satisfiability problem in first-order logic is not decidable. Similarly, the decidability of SMT($\mathcal{T}$) depends on the specific theory $\mathcal{T}$. Therefore, most state-of-the-art SMT solvers only consider decidable theories. Another typical restriction, which we will adopt within the scope of our thesis, is to permit only quantifier-free formulas φ for SMT problems and to implicitly treat all variables appearing in φ as existentially quantified.

It is the challenge of SMT solving to combine Boolean reasoning with decision procedures specific to the theory $\mathcal{T}$. While there are efficient decision procedures for many theories $\mathcal{T}$ like the Simplex algorithm [NM65] for linear arithmetic, these procedures typically only operate on conjunctions of $\mathcal{T}$-literals and therefore cannot handle arbitrary Boolean connectives. On the other hand, generalizing these procedures to support disjunctions is not trivial and usually breaks their good performance. Hence, the vast majority of SMT solvers utilizes a coupling of a SAT and a $\mathcal{T}$-solver to decide SMT($\mathcal{T}$), using an adaptation of the DPLL algorithm named DPLL(T).

The key idea behind the DPLL(T) algorithm is to use the SAT solver for assigning truth values to theory predicates. The predicates and their assigned interpretation are then passed to the $\mathcal{T}$-solver, which checks the satisfiability of their conjunction. This information is passed back to the SAT solver, which modifies its choices until the $\mathcal{T}$-solver reports the consistency of the chosen constraints. More formally, the SAT solver does not operate on the original first-order formula φ_{input}, but on its propositional skeleton:

Definition 2.1.2 (Propositional skeleton)**.** *For a first-order formula φ, its propositional skeleton* $\mathrm{skel}(\varphi)$ *is the propositional formula that is obtained from φ by replacing each first-order predicate p in φ by a propositional variable X_p.*

Example 2.1.1 (Propositional skeleton)**.** *Consider the first-order formula $\varphi : x + y = 6 \wedge (y > 2 \rightarrow x < 3)$. Its propositional skeleton is* $\mathrm{skel}(\varphi) : X_{x+y=6} \wedge (X_{y>2} \rightarrow X_{x<3})$.

A basic variant of the DPLL(T) algorithm is illustrated in Algorithm 2. Initially, the DPLL(T) function transforms the input formula into its propositional skeleton $\mathrm{skel}(\varphi)$ and starts with an empty assignment α. The functions BCP, DECIDE and RESOLVE_CONFLICT have the same semantics as in the context of the DPLL algorithm.

The transition from a propositional formula to $\mathcal{T}$-literals is performed by the T-DEDUCTION function. It creates a set of literals Φ_T from the partial assignment α according to the rule:

$$\Phi_T \;:=\; \{\, p \mid \alpha(X_p) = 1 \,\} \cup \{\, \neg p \mid \alpha(X_p) = 0 \,\} \tag{2.1}$$

Note that no literal is generated for predicate p if $\alpha(X_p)$ is still undefined. The function T-DEDUCTION then checks the consistency of the $\mathcal{T}$-literals Φ_T by applying a decision procedure for the theory $\mathcal{T}$. It returns *true* in case of consistency, and otherwise a negated conjunction of $\mathcal{T}$-literals that are a conflicting set. In the latter case, the DPLL(T) algorithm "learns" about the conflict by adding the propositional skeleton of this clause to t.

```
1:  function DPLL(T)(quantifier-free formula φ)
2:      φ ← cnf(skel(φ))
3:      if ¬BCP(φ) then                          ▷ Boolean Constraint Propagation
4:          return unsat
5:      end if
6:      while true do
7:          if ¬DECIDE(φ) then                   ▷ Make decision
8:              return sat
9:          end if
10:         repeat
11:             while ¬BCP(φ) do
12:                 if ¬RESOLVE_CONFLICT() then  ▷ Backtracking
13:                     return unsat
14:                 end if
15:             end while
16:             t ← T-DEDUCTION()
17:             t ← t ∧ skel(t)
18:         until t ≡ true
19:     end while
20: end function
```

Algorithm 2: DPLL(T)

Algorithm 2 uses a technique called *early pruning*: Instead of generating a full assignment before calling the $\mathcal{T}$-solver, the $\mathcal{T}$-consistency is already checked on the partial assignment α. As a consequence, $\mathcal{T}$-inconsistent assignments can already be detected very early, which may lead to a significant pruning of the search tree. This technique is referred to as *less lazy DPLL*, whereas the opposite strategy of calling the $\mathcal{T}$-solver only on complete assignments is named *full lazy DPLL*.

Furthermore, the method of learning invalid assignments of $\mathcal{T}$-literals by adding an appropriate clause to t is called *conflict learning* or *lemma generation*. It prevents the SAT solving engine from generating a partial assignment α containing a sub-assignment for which its $\mathcal{T}$-inconsistency has already been detected previously.

State-of-the-art implementations of DPLL(T) incorporate many optimizations in comparison to the simple version depicted in Algorithm 2. Two important techniques shall be discussed briefly:

Incrementality of $\mathcal{T}$-solver Algorithm 2 conveys the feeling that the calls to T-DEDUCTION are independent of each other and no information is shared between two invocations. As the formula set Φ_T usually only changes slightly between two consecutive calls to T-DEDUCTION, it would instead be beneficial to maintain the internal state of the $\mathcal{T}$-solver in the meantime. In this manner, plenty of redundant computations can be avoided. This functional requirement for the $\mathcal{T}$-solver is denoted by *incrementality*. In its interface, the function T-DEDUCTION is supplemented by the two functions T-ADD and T-REMOVE, which add and remove a single formula to the internal set of asserted $\mathcal{T}$-constraints.

Theory propagation By the means of *theory propagation*, the $\mathcal{T}$-solver may provide the SAT solving part with information which it has learned during a consistency check. This is limited to $\mathcal{T}$-literals in the input formula for which their truth value under α is still undefined. For example, if φ contains the variables $X_{c>1}$ and $X_{c>4}$, a solver for linear arithmetic may propagate the fact that an assignment α with $\alpha(X_{c>4}) = 1$ always needs to be completed with $\alpha(X_{c>1}) = 1$, because $c > 4 \rightarrow c > 1$ is a tautology in $\mathcal{T}$. Since this implication requires reasoning over $\mathcal{T}$, it can be seen as the $\mathcal{T}$-counterpart to Boolean Constraint Propagation.

2.1.3 SMT-LIB

By its nature, an SMT problem instance is much harder to formally specify than a SAT problem instance. The reason is that the underlying theory $\mathcal{T}$ is part of the problem and not only its domain has to be given, but also its function and predicate symbols and their respective interpretations. Every SMT solver supports a number of background theories, each with a fixed signature and fixed semantics. This way, the background theory $\mathcal{T}$ is stated in the SMT solver input only by its name. It is thereby desirable that all SMT solvers share a common understanding of the formal definition belonging to some theory name. Moreover, the syntactical structure of the input and output of SMT solvers calls for standardization, in order to allow the exchange of problems between different solvers and to enable direct comparisons in performance.

This request for standardization in the field of SMT has led to the foundation of the *SMT-LIB initiative* in 2003. Its overall mission is to establish a common ground for all global research and development in the field of SMT solving. To this end, SMT-LIB collects theories which it finds to be of common relevance, and establishes a precise definition of syntax and semantics of these theories. In addition, a universal and extendable problem description format (.smt2) is specified, which has become, together with its predecessor format (.smt), a common standard. The initiative also provides an exhaustive set of several thousands of benchmarks, grouped into different categories. On their basis, an annual competition named SMT-COMP is carried out, in which all famous SMT solvers participate on a regular basis.

Concerning the background theories, SMT-LIB further differentiates between *logics* and *theories*. Theories are the core of the specification. They define a domain and a small set of function

and predicate symbols on them. Typically, the domains of two different theories are distinct. Logics, on the other hand, can be composed of multiple theories. They can introduce extended function or predicate symbols, which are defined on the basis of the symbols originating from the theory. Moreover, logics may impose restrictions on the usage of the the theory symbols. The logic of linear integer arithmetic, for example, is based on the theory of integer numbers with the additional restriction that multiplications are only used in a linear form. A frequent restriction is the exclusion of quantifiers, denoted by the logic prefix "QF_".

At the time of writing, SMT-LIB contained a total number of 29 logics, based on the seven theories listed below:

Core A fundamental theory, included by every logic. It defines Boolean connectives, equality and inequality predicates and an if-then-else (ite) construct.

Ints Theory of integer numbers ($\mathbb{Z}$). It defines the operators $+$, $-$, $*$, *div* (integer division), *mod* (modulo) and *abs* (absolute value), as well as comparison predicates. For every $n \in \mathbb{N}$ the unary predicate $divisible_n$ is defined.

Reals Theory of real numbers ($\mathbb{R}$). It defines the operators $+$, $-$, $*$ and $/$, together with comparison predicates.

Reals_Ints Theory of integer and real numbers (as two distinct types, hence describable by the disjoint union $\mathbb{Z} \sqcup \mathbb{R}$). On integer numbers, the same operators as in the Ints theory are defined; on real numbers, the operators from the Reals theory are in effect. In addition, conversion operators between Ints and Reals exist (which is the main technical reason for introducing an own, mixed theory).

FixedSizeBitVectors Theory of bit vectors ($\bigcup_{n \in \mathbb{N}} \{ \{ 0,1 \}^n \}$). A bit vector is a tuple of elements from $\{ 0,1 \}$ with arbitrary (but positive) length. The theory defines concatenation, extraction, logical and arithmetic operators, as well as an arithmetic comparison predicate. For more details, see Section 2.3.

FloatingPoint Theory of floating point numbers. Compared to the other theories, this is the most recent and most complex theory. It serves the purpose of reasoning about machine representations of real numbers with finite precision, as they are used in calculations by most modern CPUs. Floating point sorts are indexed by the number of bits in the exponent and in the significant. Additional special values exist to represent $-\infty, \infty, +0, -0$ and *NaN*. Several calculation operators are defined, whose semantics can be controlled by a parameter determining the rounding mode to be used.

ArraysEx Theory of functional arrays. No concrete sorts are defined in this theory. Instead, for any pair of sorts S_1 and S_2, an own $array(S_1, S_2)$ sort is defined. It can be seen as a generic model for computer memory: S_1 serves as index sort, S_2 as sort for the array values. An array is built by chaining calls to a function named *store*, taking as arguments an index position and an element value. Elements can be retrieved via their index by a *select* function. The theory axioms ensure that the two functions behave as expected.

2.2 SMT-RAT

In the following, we present the SMT toolbox SMT-RAT (Satisfiability-Modulo-Theories Real Algebra Toolbox). It is an open source C++ toolbox providing support for several SMT theories.

Although SMT-RAT has its main focus on nonlinear real algebra, it also includes solving strategies for linear algebra and some non-algebraic theories like the theory of uninterpreted functions. The label "Toolbox" expresses its utilization flexibility: SMT-RAT can be composed to a full SMT solver, but it may also be integrated into foreign SMT solvers and act as a theory solver.

In a standard configuration, SMT-RAT is an incremental SMT solver based on the DPLL(T) scheme. It supports conflict-driven backjumping, early pruning, conflict learning and theory propagation. All employed $\mathcal{T}$-solvers also work in an incremental fashion. The minimalistic, but efficient and open source SAT solver MiniSat is used as underlying SAT engine.

The basic architectural component of SMT-RAT is the *module*. In most cases, a module is an implementation of some decision procedure in an SMT-compliant manner. By the term SMT-compliance, we refer to the following requirements:

- The implementation needs to be **incremental** as already addressed in Section 2.1.2. After having checked the satisfiability of a set of constraints, constraints should be addable to the module input without requiring a full new computation from scratch.
- When a conjunction of formulas is found to be unsatisfiable, the module needs to output an **infeasible subset**, i.e. an unsatisfiable subset of the input formulas. As a rule of thumb, the performance gains from conflict learning and conflict-driven backjumping are optimal if the generated unsatisfiable sets are very small or even minimal.
- The implementation must be able to perform **backtracking**, i.e. to support the belated removal of constraints from its input. Depending on the concrete definition, this property is often also subsumed under the property of incrementality.

A module is realized as a child of the SMT-RAT class `Module`. It receives its input as a list of formulas. The module instance is notified about changes in the input list by calls to its methods `add` and `remove`. Calling the `check` method starts the actual search for a solution. Here, the time-costly computations of the solving algorithm are performed. The module responds with `True` if the set of input formulas is satisfiable, `False` if it is unsatisfiable and `Unknown` if the solving method does not yield a result. The `check` method is called with a Boolean flag `_full` with the following semantics: If `_full` is *false*, the module is encouraged to skip expensive computations and to quickly fall back to the response `Unknown`. This mode is particularly suited for the early pruning approach, where a timely response is more important than a complete one. In case of the answer `True`, a model of the input formulas can be obtained by calls to `updateModel` and `model`. In case of the answer `False`, the module yields one or more infeasible subsets on the call to `infeasibleSubsets`.

In many situations, an efficient solving algorithm for some SMT theory combines several decision procedures. They often vary in terms of completeness and performance, such that it is desirable to execute incomplete, but fast decision procedures first before resorting to slow and exhaustive searches. In the terms of SMT-RAT, such a chain of modules is referred to as a *strategy*. It is represented in SMT-RAT by the class `Manager`. Formally, a strategy is defined as a rooted tree, where each node corresponds to one module. Conceptually, an edge from a module A to another module B means that A may use B for its computations, e.g. as a fallback if it cannot determine the satisfiability on its own. Module B is also called a *backend* of module A. Edges are labeled with a priority number, which is unique among the whole graph.

We now describe the technical realization of the communication between modules and their backends. For each module instance M, the set of input formulas for M is denoted by $C_{rcv}(M)$ (where *rcv* stands for *received formulas*). This input set is provided to M on its creation as

a reference to a list of formulas. M keeps a reference to this list, but does not own it. For communicating with its backends, M also manages a set $C_{pas}(M)$ of *passed formulas*. This list of formulas, which is owned by M, is passed as input to the backends of M. Formally, for every module instance B acting as backend of M, the equality $C_{rcv}(B) = C_{pas}(M)$ holds. In its `check` method, M may use B by constructing $C_{pas}(M)$ as desired and invoking its own internal method `callBackends`. This request is forwarded to the `Manager`, which lazily instantiates the backend modules of M, as defined by the strategy, and calls their `check` methods.

If a module has no backends, its internal `callBackends` call returns `Unknown`. If it has exactly one backend B, the return value of `callBackends` equals the return value of the `check` call on B. In the case of multiple backends, all backends are run sequentially until one of them returns *True* or *False* (or until all backends have returned *Unknown*). Their order is determined by the priority value of the edges of the strategy graph. Alternatively, SMT-RAT can be compiled with support for parallelization, in which case all backends are run in parallel.

In order to select different backends depending on the input formula, the edges of the strategy graph may be labeled with a condition. In this context, a condition is an arbitrary Boolean combination of formula properties. The conditions are evaluated when executing `callBackends`, and only those backends are called for which the corresponding condition is satisfied. Exemplary properties of a formula are whether it is in CNF, whether it contains inequalities or whether all arithmetic constraints are linear.

The inter-module communication concept of C_{rcv} and C_{pas} permits a very flexible usage. Three typical use cases can be identified:

Preprocessing A module M does not implement an own decision procedure, but only rewrites $C_{rcv}(M)$ such that $C_{pas}(M)$ is in a more suitable or optimized format for its backends. As an example, the `CNFerModule` brings its received formulas into CNF by applying Tseitin's transformation.

Incomplete decision procedures A module M that implements an incomplete decision procedure may use a (probably less performant) complete decision procedure as backend. In this case, the formulas are passed without modifications, i.e. $C_{rcv}(M) = C_{pas}(M)$.

Reduction A module M may implement a decision procedure for SMT($\mathcal{T}_1$) by a reduction to SMT($\mathcal{T}_2$) or SAT. Here $C_{rcv}(M)$ are FO-formulas over $\mathcal{T}_1$, whereas $C_{pas}(M)$ are FO-formulas over $\mathcal{T}_2$. Usually, the two sets are equisatisfiable.

In fact, the concept of modules and their backends is as powerful to model the structure of the DPLL(T) algorithm (see Section 2.1.2). To this end, a `SATModule` acts as a SAT solver on the propositional skeletons of its received formulas. By the strategy, the $\mathcal{T}$-solver is used as backend module. The call to T-DEDUCTION in Algorithm 2 then corresponds to the invocation of `callBackends`, where Φ_T is transported via the passed formulas of the `SATModule`.

In most cases, each formula in $C_{pas}(M)$ has one or more formulas in $C_{rcv}(M)$ from which it originates. This logical relationship is made explicit in SMT-RAT: Every formula in $C_{pas}(M)$ is stored with a set of *origins*, i.e. formulas from $C_{rcv}(M)$. Whenever a received formula is removed due to backtracking, it is also removed from all sets of origins. The set of passed formulas is then cleared of all formulas without remaining origins. This way, the module infrastructure already provides some support for incrementality.

SMT-RAT enables the user to completely customize the strategy to be used. To this end, a GUI is provided, which displays a graphical representation of the strategy graph. The GUI

allows to modify the strategy and generates the according C++ code for constructing a suitable `Manager` child class.

It is worth mentioning that the SAT module is not a mandatory part of the strategy graph. This allows for two different operation modes of SMT-RAT: With the SAT module enabled, it acts as a regular SMT solver. Typically, the SAT module appears as the root or very close to the root of the strategy graph, whereas all of its descendants are theory modules. Disabling the SAT module, on the other hand, leads to the creation of a $\mathcal{T}$-solver instead of an SMT($\mathcal{T}$) solver. In such a configuration, SMT-RAT can be embedded into foreign SMT solvers by acting as a regular theory module. Nevertheless, it can use the full functionality of the strategy graph concept and the user-defined composition of SMT-RAT modules.

For a list of modules that are currently implemented in SMT-RAT, we refer to [CKJ+15].

2.3 Bitvector Arithmetic

Modern computers are equipped with CPUs with an astonishing computational power. Millions of computations or remarkable precision can be performed within a single second. However, digital computations obey certain laws that programmers need to be aware of. Most importantly, CPUs only operate on words of a limited bit length. As soon as the representation of a value exceeds this length, the computation may yield results which seem to be unintuitive or simply wrong. Such an effect can be the reason of software errors, which are rare enough to remain undiscovered in a software testing process. Bitvector arithmetic is an SMT theory that allows for an accurate modeling of a CPU for verification purposes. It supports a rich set of operators, which resemble the instructions processed by CPUs in terms of their semantics.

2.3.1 The SMT Logic QF_BV

SMT-LIB defines the bitvector logic QF_BV. It is based on the SMT theory of fixed-size bitvectors, enriched by some convenience operators, which do not add new functionality, but simplify the usage of the theory. In the following, we present syntax and semantics of both theory and logic.

Syntax

The SMT theory of fixed-size bitvectors is based on the SMT-LIB Core theory, which defines the typical Boolean constants (`true` and `false`) and connectives (`not`, `=>`, `and`, `or`, `xor`), as well as the function symbols `=`, `distinct` and `ite`, which represent equality, inequality and if-then-else functions over arbitrary sorts. On top of the Core theory, the theory of fixed-size bitvectors introduces the following language elements:

- A sort $\texttt{BitVec}_n$ for every $n \in \mathbb{N}$
  (Technically, the indexed keyword $\texttt{BitVec}_n$ is written as `(_ BitVec n)`, but we use the simplified notation for a better readability.)
- Literals of the form `#bX`, where `X` is a sequence of `0` and `1`, and literals of the form `#xX`, where `X` is a sequence of hexadecimal characters (`0` to `F`)
- Function symbols and their signature:

  - concat : $\texttt{BitVec}_i \times \texttt{BitVec}_j \to \texttt{BitVec}_{i+j}$ $(i, j \in \mathbb{N})$
  - $\texttt{extract}_{i,j}$: $\texttt{BitVec}_m \to \texttt{BitVec}_{i-j+1}$ $(i, j \in \mathbb{N}_0, m \in \mathbb{N}, m > i \geq j)$
  - bvnot and bvneg,
    with the signature $\texttt{BitVec}_m \to \texttt{BitVec}_m$ $(m \in \mathbb{N})$
  - bvand, bvor, bvadd, bvmul, bvudiv, bvurem, bvshl and bvlshr,
    with the signature $\texttt{BitVec}_m \times \texttt{BitVec}_m \to \texttt{BitVec}_m$ $(m \in \mathbb{N})$

- The predicate symbol bvult over $\texttt{BitVec}_m \times \texttt{BitVec}_m$ $(m \in \mathbb{N})$

In addition, the theory QF_BV defines the following language elements:

- Literals of the form $\texttt{bvX}_n$, where X and n are numbers from $\mathbb{N}$ (written in base 10).
- Function symbols and their signature:
  - bvnand, bvnor, bvxor, bvxnor, bvsub, bvsdiv, bvsrem, bvsmod and bvashr,
    with the signature $\texttt{BitVec}_m \times \texttt{BitVec}_m \to \texttt{BitVec}_m$ $(m \in \mathbb{N})$
  - bvcomp : $\texttt{BitVec}_m \times \texttt{BitVec}_m \to \texttt{BitVec}_1$ $(m \in \mathbb{N})$
  - $\texttt{repeat}_i$: $\texttt{BitVec}_m \to \texttt{BitVec}_{i \cdot m}$ $(i, m \in \mathbb{N})$
  - $\texttt{zero\_extend}_i$ and $\texttt{sign\_extend}_i$,
    with the signature $\texttt{BitVec}_m \to \texttt{BitVec}_{m+i}$ $(m \in \mathbb{N}, i \in \mathbb{N}_0)$
  - $\texttt{rotate\_left}_i$ and $\texttt{rotate\_right}_i$,
    with the signature $\texttt{BitVec}_m \to \texttt{BitVec}_m$ $(m \in \mathbb{N}, i \in \mathbb{N}_0)$
- The predicate symbols bvule, bvugt, bvuge, bvslt, bvsle, bvsgt and bvsge over $\texttt{BitVec}_m \times \texttt{BitVec}_m$ $(m \in \mathbb{N})$

It should be noted that in SMT-LIB, function and predicate applications are written using the syntax `(bvand x y)`. Throughout the thesis, we use the more familiar notation `bvand(x,y)`.

Semantics

A bitvector of length n represents an ordered sequence of n bits (0 or 1). Different formalizations are possible. While it is possible to represent an n-bit bitvector as an n-tuple from $\{0,1\}^n$, a formalization based on functions is often handier in practice. Here, an n-bit bitvector is described by a function $f : \{0, \ldots, n-1\} \to \{0,1\}$, which maps every position to its value. For a shorter notation, we use a lambda expression as defined by Church [Chu85], such that we can write $\lambda i \in \{0, \ldots, n-1\}.f(i)$ instead. Note that $i = 0$ refers to the last bit in the sequence (the *least significant bit*), whereas $i = n-1$ is the first bit in the sequence (*most significant bit*).

Example 2.3.1. *The bitvector* $\lambda i \in \{0, \ldots, 7\}.(1$ *if* $i \leq 3$, *else* $0)$ *has the value* `00001111`. *Using the ternary operator "condition ? then : else" (as found in many programming languages), we can also write* $\lambda i \in \{0, \ldots, 7\}.(i \leq 3)\ ?\ 1 : 0$.

The i-th bit of a bitvector b is the function value $b(i)$. In the following, we also permit the abbreviation b_i. The set of all bitvectors of length n is denoted by $BVec_n$.

A frequently encountered purpose of bitvectors is the storage of integer numbers. To this end, a mapping between a set $N \subset \mathbb{N}$ of natural numbers and the set $BVec_n$ has to be fixed for each $n \in \mathbb{N}$. Such a mapping is also referred to as *encoding*. In practice, two different encodings are used, depending on whether N should include negative numbers or not.

Definition 2.3.1 (*nat2bv* and *bv2nat*)**.** *For each $n \in \mathbb{N}$, the function $bv2nat_n : BVec_n \to \{0, 1, \ldots, 2^n - 1\}$ is defined by: $bv2nat_n(b) := \sum_{i=0}^{n-1} b_i \cdot 2^i$.*

The function $nat2bv_n : \{0, 1, \ldots, 2^n - 1\} \to BVec_n$ is defined via $nat2bv_n := bv2nat_n^{-1}$.

Example 2.3.2. *For $n = 8$ we have $bv2nat_n(\texttt{00000000}) = 0$, $bv2nat_n(\texttt{00101010}) = 42$, $bv2nat_n(\texttt{10000000}) = 128$ and $bv2nat_n(\texttt{11111111}) = 255$.*

By this definition, *nat2bv* and *bv2nat* behave like a computer storing integer values of an *unsigned* type. The expression $nat2bv_n(m)$ creates the representation of m in base 2, using exactly n digits. However, there is no canonical representation of negative integer numbers, which makes it unclear how *signed* values should be expressed. Nevertheless, one encoding scheme has become the quasi-standard nowadays due to some properties which facilitate their use for calculating. It is called *two's complement*:

Definition 2.3.2 (*int2bv* and *bv2int*)**.** *For each $n \in \mathbb{N}$, the function $bv2int_n : BVec_n \to \{-2^{n-1}, -2^{n-1} + 1, \ldots, 2^{n-1} - 1\}$ is defined by: $bv2int_n(b) := -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$. The function $int2bv_n : \{-2^{n-1}, -2^{n-1} + 1, \ldots, 2^{n-1} - 1\} \to BVec_n$ is defined as its inverse function $int2bv_n := bv2int_n^{-1}$.*

Example 2.3.3. *For $n = 8$ we have $bv2int_n(\texttt{00000000}) = 0$, $bv2int_n(\texttt{00101010}) = 42$, $bv2int_n(\texttt{10000000}) = -128$ and $bv2int_n(\texttt{11111111}) = -1$.*

In QF_BV, bitvector literals can be written in three ways. Consider, for example, the bitvector `11000101`. It has a binary representation (`#b11000101`, from most-significant to least-significant bit), a hexadecimal representation (`#xC5`, only admissible for lengths being a multiple of 4) and a decimal representation ($\texttt{bv197}_8$, corresponding to the function value $nat2bv_8(197)$).

We now define the semantics by giving the interpretation of all function symbols. If no function signature is stated explicitly, the signature is $BVec_m \times BVec_m \to BVec_m$ (for $m \in \mathbb{N}$).

Core symbols On bitvectors, the predicate symbols = and `distinct` have the semantics of the following relations bveq and bvneq:

$$\begin{aligned} \text{bveq}, \text{bvneq} &\subseteq BVec_m \times BVec_m \ (m \in \mathbb{N}) \\ \text{bveq} &= \{ (b, b) \mid b \in BVec_m \} \\ \text{bvneq} &= (BVec_m \times BVec_m) \setminus \text{bveq} \end{aligned}$$

The ternary operator `ite` on bitvectors can be defined as follows:

$$\begin{aligned} \text{ite} &: \{0, 1\} \times BVec_m \times BVec_m \to BVec_m \\ \text{ite}(i, t, e) &:= \begin{cases} t & \text{if } i = 1 \\ e & \text{if } i = 0 \end{cases} \end{aligned}$$

Bitwise operators The only unary bitwise operator `bvnot`, performs bitwise negation. Each bit of the input is flipped:

$$\begin{aligned} \text{bvnot} &: BVec_m \to BVec_m \ (m \in \mathbb{N}) \\ \text{bvnot}(b) &:= \lambda i \in \{0, \ldots, m - 1\}.\neg b_i \end{aligned}$$

All binary bitwise operators (`bvand`, `bvor`, `bvnand`, `bvnor`, `bvxor`, `bvxnor`) apply the corresponding binary function bit by bit. As an example, `bvnand` has the semantics:

$$\text{bvnand}(a,b) := \lambda i \in \{\,0,\ldots,m-1\,\}.\neg(a_i \wedge b_i)$$

A special role has `bvcomp` ("compare"). It compares its inputs for equality and outputs the result as a bitvector of length 1:

$$\text{bvcomp} : BVec_m \times BVec_m \to BVec_1 \ (m \in \mathbb{N})$$
$$\text{bvcomp}(a,b) := \begin{cases} \lambda i \in \{\,0\,\}.1 & \text{if } a = b \\ \lambda i \in \{\,0\,\}.0 & \text{otherwise} \end{cases}$$

Extraction and Concatenation The operator `concat` ("concatenate") appends the second bitvector to the first one:

$$\text{concat} : BVec_m \times BVec_n \to BVec_{m+n} \ (m,n \in \mathbb{N})$$
$$\text{concat}(a,b) := \lambda i \in \{\,0,\ldots,m+n-1\,\}.(i < m)\ ?\ a_i : b_{i-m}$$

A part of a bitvector can be obtained using $\texttt{extract}_{i,j}$. Here, i is the index of the first (most-significant) bit to be extracted, and j the index of the last (least-significant) one. For this reason, the constraint $i \geq j$ is needed:

$$\text{extract}_{i,j} : BVec_m \to BVec_{i-j+1} \ (i,j \in \mathbb{N}_0, m \in \mathbb{N}, m > i \geq j)$$
$$\text{extract}_{i,j}(b) := \lambda k \in \{\,0,\ldots,i-j\,\}.b_{j+k}$$

The operator $\texttt{repeat}_i$ is a shortcut for multiple applications of `concat`.

$$\text{repeat}_i : BVec_m \to BVec_{i \cdot m} \ (i,m \in \mathbb{N})$$
$$\text{repeat}_1(b) := b$$
$$\text{repeat}_i(b) := \text{concat}(b, \text{repeat}_{i-1}(b)) \ (\text{for } i > 1)$$

Extension Bitvectors can be extended by a non-negative amount of bits, leaving their numeric value unchanged. Two separate variants exist for unsigned and signed extension:

$$\text{zero\_extend}_i, \text{sign\_extend}_i : BVec_m \to BVec_{m+i} \ (m \in \mathbb{N}, i \in \mathbb{N}_0)$$
$$\text{zero\_extend}_i(b) := \text{nat2bv}_{m+i}(\text{bv2nat}_m(b))$$
$$\text{sign\_extend}_i(b) := \text{int2bv}_{m+i}(\text{bv2int}_m(b))$$

Shifting and Rotation Shifting operators move the bits of the first bitvector to the left or the right, by as many steps as given by the value of the second bitvector. In the case of the operators `bvshl` ("shift left") and `bvlshr` ("logical shift right"), the bitvector is filled up with zeros from the right or left side, respectively. The operator `bvashr` ("arithmetic shift right") fills up from the left with replications of the most-significant bit of the first bitvector.

$$\text{bvshl}(a,b) := \lambda i \in \{\,0,\ldots,m-1\,\}.\begin{cases} a_{i-\text{bv2nat}_m(b)} & \text{if } i - \text{bv2nat}_m(b) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{bvlshr}(a,b) := \lambda i \in \{\,0,\ldots,m-1\,\}.\begin{cases} a_{i+\text{bv2nat}_m(b)} & \text{if } i + \text{bv2nat}_m(b) < m \\ 0 & \text{otherwise} \end{cases}$$

$$\text{bvashr}(a,b) := \lambda i \in \{\,0,\ldots,m-1\,\}.\begin{cases} a_{i+\text{bv2nat}_m(b)} & \text{if } i + \text{bv2nat}_m(b) < m \\ a_{m-1} & \text{otherwise} \end{cases}$$

The functions `rotate_left`$_i$ and `rotate_right`$_i$ have a similar behavior to the shifts. They also move the bits of the bitvector by i steps to the left or the right. In contrast to shifting, the left- or rightmost bits are not discarded, but replicated at the other end:

$$\begin{aligned}
&\text{rotate_left}_i : BVec_m \to BVec_m \ (m \in \mathbb{N}, i \in \mathbb{N}_0) \\
&\text{rotate_left}_0(b) := b \\
&\text{rotate_left}_i(b) := \lambda k \in \{\, 0, \ldots, m-1 \,\}.\, \begin{cases} (\text{rotate_left}_{i-1}(b))_{m-1} & \text{if } k = 0 \\ (\text{rotate_left}_{i-1}(b))_{k-1} & \text{otherwise} \end{cases} \quad (\text{for } i > 0) \\
&\text{rotate_right}_i := \text{rotate_left}_i^{-1}
\end{aligned}$$

Arithmetic Bitvectors of length m can represent values between 0 and $2^m - 1$ using unsigned encoding, and values between -2^{m-1} and $2^{m-1} - 1$ using signed encoding. If a numeric value exceeds this range, it cannot be represented any more. In case of an addition of the unsigned values $\text{nat2bv}_8(200) =$ `11001000` and $\text{nat2bv}_8(100) =$ `01100100`, for example, an addition yields `00101100` (the 8 least significant bits of `100101100`), which corresponds to the value 44 instead of 300. Hence, bitvector arithmetic uses *modular arithmetic*, since it computes correct results modulo 2^m ($300 \equiv 44 \bmod 2^8$). Based on this observation, the semantics of the arithmetic operations can easily be defined:

$$\begin{aligned}
\text{bvneg}(b) = c &\iff \text{bv2nat}_m(c) \equiv -\text{bv2nat}_m(b) &&(\bmod 2^m) \\
\text{bvadd}(a,b) = c &\iff \text{bv2nat}_m(c) \equiv \text{bv2nat}_m(a) + \text{bv2nat}_m(b) &&(\bmod 2^m) \\
\text{bvsub}(a,b) = c &\iff \text{bv2nat}_m(c) \equiv \text{bv2nat}_m(a) - \text{bv2nat}_m(b) &&(\bmod 2^m) \\
\text{bvmul}(a,b) = c &\iff \text{bv2nat}_m(c) \equiv \text{bv2nat}_m(a) \cdot \text{bv2nat}_m(b) &&(\bmod 2^m)
\end{aligned}$$

Due to the employment of two's complement, there is no need for a signed variant of the above operators, because they would behave exactly like their unsigned counterparts. This does not hold for the operators of integer division and remainder. They are defined as follows for $\text{nat2bv}_m(b) \neq 0$:

$$\begin{aligned}
\text{bvudiv}(a,b) &= \text{nat2bv}_m(\max\{\, c \in \mathbb{N}_0 : c \cdot \text{bv2nat}_m(b) \leq \text{bv2nat}_m(a) \,\}) \\
\text{bvsdiv}(a,b) &= \text{int2bv}_m(\max\{\, c \in \mathbb{Z} : c \cdot \text{bv2int}_m(b) \leq \text{bv2int}_m(a) \,\}) \\
\text{bvurem}(a,b) &= \text{nat2bv}_m(\text{bv2nat}_m(a) - \text{bv2nat}_m(b) \cdot \text{bv2nat}_m(\text{bvudiv}(a,b))) \\
\text{bvsrem}(a,b) &= \text{nat2bv}_m(\text{bv2int}_m(a) - \text{bv2int}_m(b) \cdot \text{bv2int}_m(\text{bvsdiv}(a,b))) \\
\text{bvsmod}(a,b) &= \begin{cases} \text{bvadd}(\text{bvsdiv}(a,b), b) & \text{if } \text{bv2int}_m(b) < 0 \text{ and } \text{bv2nat}_m(\text{bvsdiv}(a,b)) \neq 0 \\ \text{bvsdiv}(a,b) & \text{otherwise} \end{cases}
\end{aligned}$$

For $\text{nat2bv}_m(b) = 0$, the above expressions are not defined. Hence, an SMT solver may not make any assumptions about their value, and instead has to assume that these expressions may take an arbitrary value in the theory.

The semantics of the predicate symbols are straightforward. As an example, consider the predicate `bvule` ("unsigned less or equal"):

$$\begin{aligned}
&\text{bvule} \subseteq BVec_m \times BVec_m \ (m \in \mathbb{N}) \\
&\text{bvule} = \{\, (a,b) \in BVec_m \times BVec_m : \text{bv2nat}_m(a) \leq \text{bv2nat}_m(b) \,\}
\end{aligned}$$

The remaining predicate symbols `bvult` ("unsigned less than"), `bvugt` ("unsigned greater than"), `bvuge` ("unsigned greater or equal") and their signed equivalents can easily be defined analogously. To this end, the signed predicates use bv2int instead of bv2nat.

2.3.2 Decision Procedures

The decision procedures which have been implemented for bitvector arithmetic can be divided into three groups. For each group, we present the approach, refer to a number of representative implementations and briefly evaluate the strengths and weaknesses of the respective procedure.

Canonizer & Solver

The earliest relevant approach goes back to Shostak [Sho82], who gives a general algorithm for deciding combinations of theories which fulfill certain properties. Being published in 1982, this concept emerged at a time long before the term SMT came up and the DPLL(T) algorithm became popular. The method described by Shostak handles σ-theories, i.e. theories $\mathcal{T}$ for which a *canonizer* function exists. A canonizer maps $\mathcal{T}$-terms to $\mathcal{T}$-terms, generating a normal form for every fully interpreted term. The biggest restriction to $\mathcal{T}$ is that it also needs to be *algebraically solvable*, i.e. that a solving function must exist which takes a set of canonized equations and generates canonized solutions for the contained variables.

Such a combination of a canonizer and a solver for bitvector arithmetic is suggested in [MR98]. It was implemented in the SMT solver ICS (Integrated Canonizer and Solver) [FORS03], which is no longer under active development. A similar work is [BDL98], which describes the implementation of a Shostak-based decision procedure for bitvector arithmetic in the Stanford Validity Checker [BDL96]. Both variants, however, are restricted to a small fraction of bitvector arithmetic as defined by SMT-LIB. Only combinations of concatenation, extraction, equality and linear equations can be handled and an extension to further operators seems to be difficult. ICS has been superseded by the Yices solver [DDM06] and SVC has developed into CVC [SBD02], both of which replaced the Shostak-based decision procedure by a SAT-driven approach, as explained in the following.

Encoding to SAT

The most prominent approach of deciding bitvector arithmetic nowadays is a reduction to the SAT problem, also known as *bit-blasting* or *flattening*. To this end, each bitvector variable of width n is replaced by n propositional variables. All bitvector terms and constraints are then expressed on the basis of these variables, much similar to electronic circuits found in CPUs. A SAT solver decides about the satisfiability of the equisatisfiable SAT instance. By re-composing the propositional variables to the original bitvectors, a model for the SAT instance can be directly rewritten into a model for the original bitvector instance. For a more detailed and more technical description of bit-blasting, see Section 3.2 about our implementation in SMT-RAT.

Two different bit-blasting strategies can be distinguished: The first strategy is to encode all bitvector constraints upfront before applying the SAT solver for the first time. Alternatively, one can start with a SAT search over the propositional skeleton of the original formula, just as in the DPLL(T) algorithm, and add the flattened bitvector constraints only on demand. These two variants are called *eager* and *lazy flattening*. Eager flattening has the advantage that the Boolean reasoning of the SAT solver can work over the full formula and that only a single run

of the SAT solver is required. Lazy flattening, on the other hand, may show its benefits when being applied to a formula that consists of many arithmetic bitvector operations, which usually require a quite great amount of propositional constraints to be encoded to. The SAT solver is forced to reason about the high-level Boolean structure first, before diving into the circuits for each operation. Depending on the formula, this may avoid the need to encode every bitvector constraint, and prevent unnecessary searches over irrelevant propositional formulas.

Boolector [BB09], the winner of the QF_BV division in the SMT-COMP of 2015 and 2014, uses bit-blasting on bitvector formulas. It further employs refinement loops and under-approximations, which add further constraints on the CNF level to reduce the search space. A similar idea of an iterative refinement of abstractions is described in [BKO+07] and implemented in UCLID [LS04]. CVC4 [BCD+11], a descendant of CVC [SBD02], applies lazy bit-blasting after simple preprocessing steps. STP [GD07] uses substitution, simplification and linear solving steps before eagerly encoding to SAT. Yices [DDM06] and Z3 [DMB08] flatten all bitvector operations except for equality in a DPLL(T) framework. In MathSAT [CGSS13], the bitvector constraints are first checked for unsatisfiability by an EUF-solver before resorting to bit-blasting. To summarize, all participants in the QF_BV division in the SMT-COMP of the last years are based on reductions to SAT, which is a clear indicator for the success of this approach.

Encoding to Integer Arithmetic

bit-blasting has its biggest deficiencies in the handling of arithmetic bitvector operations. In particular, multiplication circuits quickly become large as the bitvector width increases, leading to the introduction of numerous propositional variables and a blowup of the Boolean search space. This gives rise to the idea of reducing to Integer Arithmetic instead of SAT, such that the benefits of an arithmetic solver can be exploited. Nevertheless, due to the modular semantics in bitvector arithmetic, even the translation of the bitvector operators addition and multiplication is not trivial.

Brinkmann and Drechsler [BD02] presented an encoding of a fragment of bitvector arithmetic into an Integer Linear Program in the context of RTL descriptions of hardware circuits. The ILP is then decided using the Omega test [Pug91]. Parthasarathy et al. [PICW04] base their work on this approach, adding a DPLL(T) structure to additionally support Boolean operators, which are not considered in [BD02]. Both implementations are restricted to linear bitvector operators. Similarly, Huang and Cheng [HC00] apply a solver for linear modular arithmetic to solve bitvector equations. Nonlinear operators are only supported by heuristical enumeration and substitution of possible solutions in order to fall back to linear arithmetic. Most recently, Babić and Musuvathi [BM05] employed Hensel's Lemma, a generalization of the Newton method to p-adic case, to handle nonlinear operations more efficiently. However, the concept of encoding to linear or nonlinear Integer arithmetic seems to be very limited and hard to extend to the full set of bitvector operations as defined in SMT-LIB.

2.4 Integer Arithmetic

While the theory of bitvectors clearly originates from the field of verification and has not been of any interest before the emergence of computers, the theory of integer arithmetic already gained attraction by Hilbert in 1900. It took decades until a final answer to Hilbert's Tenth Problem was found, which addressed a question of decidability of integer arithmetic. Although this theory can be formulated much easier than the rich bitvector arithmetic, it is still much harder to handle.

Before we elaborate on modern solving methods, we present integer arithmetic as it is defined by the SMT-LIB standard.

2.4.1 The SMT Logics QF_LIA and QF_NIA

SMT-LIB defines the two logics QF_LIA (Quantifier-free fragment of Linear Integer Arithmetic) and QF_NIA (Quantifier-free fragment of Nonlinear Integer Arithmetic). Both logics are based on the SMT-LIB theory "Ints" of integer numbers. Its domain is the set $\mathbb{Z}$, on which the following functions and relations are defined:

- The unary negation function (−)
- The binary subtraction, addition and multiplication functions (−, + and $\star$)
- The binary functions `div` and `mod`
- The unary function `abs`
- The unary relation $\texttt{divisible}_n$ (for $n \in \mathbb{N}$)
- The binary relations <=, <, >= and >

The semantics of most function and relation symbols are obvious. The functions div and mod are defined by the two constraints $(n \cdot \mathrm{div}(m,n)) + \mathrm{mod}(m,n) = m$ and $0 \leq \mathrm{mod}(m,n) < |n|$ for all $n \in \mathbb{Z}, m \in \mathbb{Z} \setminus \{0\}$. Considering the case $m = 0$, the interpretation of `mod(m,n)` and `div(m,n)` is not prescribed, and must be assumed to be some unknown value (i.e., the theory is underspecified for this case). The function symbol `abs` is interpreted by the absolute value function $\mathrm{abs} : \mathbb{Z} \to \mathbb{Z}, n \mapsto |n|$. For $n \in \mathbb{N}$, the relation $\mathrm{divisible}_n$ is given by $\mathrm{divisible}_n := \{ an \mid a \in \mathbb{Z} \}$.

The logic QF_NIA is defined as the theory "Ints" plus arbitrary free constant symbols (what we refer to as existentially quantified variables). The sublogic QF_LIA is derived from QF_NIA by imposing the additional restriction that all integer terms must be linear, i.e. that the formulas do not use the function symbols $\star$, `div`, `mod` and `abs`. The only permitted use of $\star$ is in combination with a constant integer argument (which is only syntactical sugar, as such a product can be replaced by an equivalent sum).

In the context of our thesis, we ignore the function symbols `div`, `mod` and `abs` as well as the relation symbols $\texttt{divisible}_n$. Without these symbols, the set of constraints expressible in QF_NIA (or QF_LIA) is exactly the set of (linear) Diophantine constraints. Each such constraint can be described by a multivariate polynomial being compared to zero, which simplifies the internal representation, its canonization and the algorithms applied on it.

2.4.2 Decision Procedures

Nowadays, problems in linear arithmetic over the reals can be solved very efficiently. Most prominently, the Simplex algorithm is a technique for solving Linear Programs. Despite its exponential worst-case complexity, Simplex has turned out to be very fast on practically relevant instances. Unfortunately, Simplex alone is not sufficient for solving integer linear arithmetic because it may generate assignments containing non-integer values. Moreover, the problem of deciding a conjunction of linear Diophantine constraints has been proven to be NP-complete, in

contrast to its non-integer counterpart, which is in P. Hence, we cannot expect any algorithms for linear integer arithmetic which have a polynomial worst-case complexity. Most implementations today are based on the following techniques:

Refinement of LP-relaxation Given an integer linear program (ILP), its LP-relaxation is the problem consisting of the same constraints, without the requirement that all solutions must be integer. The LP-relaxation is solved by an algorithm for linear programs like the Simplex algorithm. If the relaxation is unsatisfiable, the ILP is unsatisfiable as well. Otherwise a solution for the LP-relaxation is returned. If it contains non-integer values, the LP-relaxation is refined to exclude the previously found solution from the new solution space. The process is repeated until the relaxation is unsatisfiable or an integer solution is returned. For the refinement step, typical approaches are Branch & Bound and cutting plane methods [Gom63].

Omega test The Omega test [Pug91] is an adaptation of the Fourier-Motzkin variable elimination scheme for integer constraints. It is based on the idea to iteratively select a variable and project all of its constraints to the remaining variables. By this step, a variable is eliminated (at the cost of potentially more constraints). The iteration ends as soon as only one variable is left and the problem can easily be decided.

Automata theory An automata-theoretic decision procedure has been suggested [GBD02], which reduces the problem of linear integer arithmetic to the emptiness problem on deterministic finite automata. To this end, a construction is described to create an automaton from a set of constraints which accepts the language of binary strings representing a solution to the ILP. Unfortunately, many of the constructs scale badly and seem to be infeasible for a practical application.

The nonlinear analogon to ILPs was already mentioned by Hilbert as the 10th item of his famous list of mathematical problems [Hil00] from 1900. He asks for a decision algorithm for the satisfiability problem over Diophantine constraints. It took 70 years until this question finally received a negative answer [Mat70]: No such algorithm can be found because the problem is undecidable. Despite this discouraging result, some (necessarily incomplete) algorithms exist which can solve some of the problems of nonlinear integer arithmetic:

Cylindrical Algebraic Decomposition The algorithm of cylindrical algebraic decomposition has its main application in nonlinear real arithmetic. It divides the search space of $\mathbb{R}^n$ into so-called *cells* with the property that all polynomials in the input constraints have constant sign on each of these cells. In the real-valued case, satisfiability can then be decided by regarding each cell on its own. When applied to integer arithmetic, the algorithm further has to enumerate all integer values that reside in every cell, which may lead to a huge number of test points.

Gröbner Basis Construction The concept of a Gröbner Basis originates from commutative algebra and algebraic geometry. A Gröbner Basis is a special finite generating set of an ideal, which can be computed by Buchberger's algorithm [Buc85]. In the context of nonlinear integer programming, the construction of a Gröbner Basis can prove the unsatisfiability of the program.

Virtual Substitution In the case that some variables only appear with a low degree in the input polynomials, the Virtual Substitution method [Wei98] can be applied. It computes terms for the roots of some polynomials and substitutes them into the remaining constraints,

such that the original variable is eliminated. An tailored version for SMT is described in [CÁ11]. The method is not specific to integer arithmetic, but also applies to nonlinear real arithmetic.

Encoding to SAT AProVE [FGM+07], the winner of the 2015 SMT-COMP in the QF_NIA division, encodes the Diophantine constraints into SAT, restricting the range of all variables to $\{0, \ldots, 2^k - 1\}$ for a fixed k. Z3 [DMB08] follows a similar approach, but it iteratively increases k if the SAT solver detects unsatisfiability. Naturally, due to the introduction of bounds, this approach can in most cases only detect the satisfiability and not the unsatisfiability of the input problem.

Linearization Instead of a reduction to SAT, the authors of [BLNM+09] propose an encoding of non-linear integer constraints to linear integer constraints by instantiating one factor of every non-linear product with values inside certain bounds.

ICP and Testing The raSAT solver [TO13] searches for a solution by alternately applying Interval Constraint Propagation, testing and decomposition. Unsatisfiability of the problem may be detected by Interval Constraint Propagation, while satisfiability may be detected by testing. If none of both events occur, the input intervals are decomposed into smaller intervals, which then serve as the starting point for the next iteration.

Finite Domain Constraint Satisfaction Search In [CMTU05], Contejean et al. propose a method for solving Diophantine equations in the setting that each variable ranges over a finite domain. The main algorithm consists of a repeated propagation of interval constraints for each variable, followed by the choice of a single variable and the isolation of the minimum or maximum of its interval. The algorithm is implemented in the termination prover CiME [CMMU03].

Among the previously presented approaches, SMT-RAT currently supports Simplex, Fourier-Motzkin, Virtual Substitution, Gröbner Basis construction and Cylindrical Algebraic Decomposition. It further provides a module for Interval Constraint Propagation, which we use as a component to develop a SAT-based module for nonlinear integer arithmetic.

Chapter 3

Solving Bitvector Arithmetic

This chapter is devoted to our implementation of a module for bitvector arithmetic in the SMT-RAT solver, which we named `BVModule`. We first describe the general composition of the module and its usage in Section 3.1. The module performs an encoding into SAT, which is presented in more detail in Section 3.2. We conclude the chapter with Section 3.3 about various optimizations which we employed for an improved efficiency.

3.1 BVModule Overview

The `BVModule` is designed to handle arbitrary input formulas. All bitvector-related constraints are removed from the input and replaced with an encoding into SAT. The encoded formulas and all input formulas without bitvector constraints are passed to the backend, which then decides the satisfiability on the propositional level.

With this behavior, the `BVModule` can be embedded into a strategy for SMT(QF_BV) as shown in Figure 3.1. The encoded propositional formulas are transformed into Conjunctive Normal Form by the `CNFModule`, before the satisfiability is ultimately decided by the SAT solver provided by the `SATModule`.

Before the strategy defined by the strategy graph is executed, the input formulas are parsed into the internal formula representation. In this process, two transformations are made which are also relevant for bitvector formulas. Firstly, wherever SMT-LIB allows syntactic sugar in the form of expressions like $\text{and}(a, b, c)$ as an abbreviation for $\text{and}(\text{and}(a, b), c)$, these compact expressions are replaced with their expanded equivalents. Secondly, all if-then-else-expressions are eliminated by rewriting each instance $\text{ite}(\mathit{if}, \mathit{then}, \mathit{else})$ into a fresh variable V with the two additional constraints $\mathit{if} \rightarrow (V = \mathit{then})$ and $\neg \mathit{if} \rightarrow (V = \mathit{else})$. In the following, we can thereby expect all formulas to be free of ite expressions.

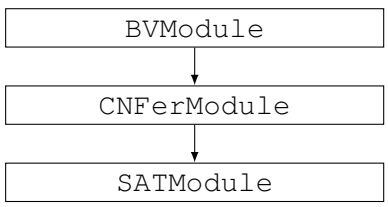


Figure 3.1: A strategy graph for SMT(QF_BV)

In order to express bitvector constraints on the propositional level, each bitvector term of width n in the input formulas is represented by a vector of n propositional variables. Similarly, each bitvector atom in the input formulas is represented by a single propositional variable. This relationship between a bitvector expression and its Boolean variables is formalized by a *bitvector mapping*:

Definition 3.1.1 (Bitvector mapping)**.** *A bitvector mapping is a mapping f, whose domain* $\mathrm{dom}(f)$ *consists of bitvector terms and atoms. For a bitvector atom $a \in \mathrm{dom}(f)$, $f(a)$ is a single propositional variable. For a bitvector term $t \in \mathrm{dom}(f)$ of width n, $f(t)$ is an n-elementary vector of propositional variables.*
The elements of $f(t)$ are referred to as $f(t)_0, \ldots, f(t)_{n-1}$. $V(f)$ denotes the set of all variables occurring in the image of f. We call f injective if $f(x)$ and $f(y)$ share no variables for any $x, y \in \mathrm{dom}(f)$.

Algorithm 3 depicts the decision algorithm implemented by the `BVModule`. Here the sets $BVAtoms(\varphi)$ and $BVTerms(\varphi)$ are the sets of all bitvector atoms or terms, respectively, appearing in φ. The algorithm starts by invoking the function freshBitvectorMapping. It creates an injective bitvector mapping e over all bitvector atoms and terms (including their subterms) appearing in C_{rcv} by choosing fresh variables for each function value of e.

```
1: function BVModule.decide(input formulas C_rcv)
2:     e ← freshBitvectorMapping(C_rcv)
3:     C_pas ← ∅
4:     for all φ ∈ C_rcv do
5:         C_pas ← C_pas ∪ { skeleton(e, φ) }
6:         for all a ∈ BVAtoms(φ) do
7:             C_pas ← C_pas ∪ BVconstraints(e, a)
8:         end for
9:         for all t ∈ BVTerms(φ) do
10:            C_pas ← C_pas ∪ BVconstraints(e, t)
11:        end for
12:    end for
13:    return backend.decide(C_pas)
14: end function
```

Algorithm 3: BVModule.decide

On the basis of e, all bitvector constraints in the input formulas are rewritten into propositional logic. As a first step, the function call $\textsc{skeleton}(e, \varphi)$ substitutes each bitvector atom a in φ by its corresponding variable $e(a)$. This substitution is in analogy to the construction in Definition 2.1.2. Note, however, that our implementation allows the formulas C_{rcv} to contain predicate symbols from other theories, which might be handled by subsequent modules in the strategy. All non-bitvector predicates and functions are left unmodified by the call to skeleton.

The key component of the `BVModule` is the function BVConstraints, which takes as argument the bitvector mapping e and a bitvector expression b (i.e., a bitvector atom or term). It returns a set of constraints Φ which emulate the semantics of b on the propositional level. To put it differently, in every model of Φ the values for $e(b)$ are computed correctly from the values for its direct subterms. For a more formal definition and a construction of BVConstraints, we refer to the following Section 3.2.

After calling the BVConstraints function for each bitvector atom and term, the formula skeletons and all generated constraints are added to the set C_{pas} of passed formulas, which is

checked for satisfiability by the backend module.

3.2 Encoding to SAT

Before we give a constructive description of the BVConstraints method, we establish its functional requirements in the following Section 3.2.1. In the remaining parts of Section 3.2, the generated constraints of BVConstraints(e, a) and BVConstraints(e, t) are presented for different terms t and atoms a.

3.2.1 Functional Requirements

A bitvector mapping e describes a relationship between bitvector expressions and propositional variables. It enables us to compare interpretations for bitvectors to interpretations for Boolean variables. Consider, for example, a bitvector variable $b_{[4]}$, and $e(b) = \langle B_3, B_2, B_1, B_0 \rangle$. Then we consider an interpretation $\mathfrak{I}_1$ which assigns $\mathfrak{I}_1(b) = \texttt{0010}$ to be equivalent to an interpretation $\mathfrak{I}_2$ with $\mathfrak{I}_2(B_3) = \mathfrak{I}_2(B_2) = \mathfrak{I}_2(B_0) = 0$ and $\mathfrak{I}_2(B_1) = 1$. In the following, we define the notion of equivalence of interpretations under a bitvector mapping more formally.

Definition 3.2.1. *Let $\mathfrak{I}_1$, $\mathfrak{I}_2$ be two interpretations, e a bitvector mapping, $t_{[n]} \in \mathrm{dom}(e)$ a bitvector term and $a \in \mathrm{dom}(e)$ a bitvector atom.*

We write $\mathfrak{I}_1 \equiv_e^t \mathfrak{I}_2$ if $(\mathfrak{I}_1(t))_i = \mathfrak{I}_2(e(t)_i)$ for each $i \in \{0, \ldots, n-1\}$.

We further write $\mathfrak{I}_1 \equiv_e^a \mathfrak{I}_2$ if it holds that $\mathfrak{I}_1 \models a$ if and only if $\mathfrak{I}_2 \models e(a)$.

Intuitively, $\mathfrak{I}_1 \equiv_e^t \mathfrak{I}_2$ means that the term t is interpreted by equivalent values in $\mathfrak{I}_1$ and $\mathfrak{I}_2$ with respect to the bitvector mapping e, where the bitvectors in $\mathfrak{I}_1$ are replaced with their corresponding Boolean variables in $\mathfrak{I}_2$.

Now the desired semantics of BVConstraints can be described as follows. Let e be an injective bitvector mapping. Then BVConstraints has to respect the following rules:

Satisfiability under all variable assignments. Let $V \subseteq \mathrm{dom}(e)$ be the set of all bitvector variables which are an element of $\mathrm{dom}(e)$. Given any suitable interpretation $\mathfrak{I}^{BV}$ for V, a model $\mathfrak{I}^P$ of $\bigcup_{b \in \mathrm{dom}(e)} \text{BVConstraints}(e, b)$ exists with $\mathfrak{I}^{BV} \equiv_e^v \mathfrak{I}^P$ for all $v \in V$.

Correct semantics of terms. Let $t = f(t_1, \ldots, t_n) \in \mathrm{dom}(e)$ be a bitvector term. Let $\mathfrak{I}^{BV}$ be a suitable interpretation for t, and let $\mathfrak{I}^P$ be a model of $\Phi = \text{BVConstraints}(e, t)$, such that $\mathfrak{I}^{BV} \equiv_e^{t_i} \mathfrak{I}^P$ for all $1 \leq i \leq n$. Then $\mathfrak{I}^{BV} \equiv_e^t \mathfrak{I}^P$ holds.

Correct semantics of atoms. Let $a = p(t_1, \ldots, t_n) \in \mathrm{dom}(e)$ be a bitvector atom. Let $\mathfrak{I}^{BV}$ be a suitable interpretation for a, and let $\mathfrak{I}^P$ be a model of $\Phi = \text{BVConstraints}(e, a)$, such that $\mathfrak{I}^{BV} \equiv_e^{t_i} \mathfrak{I}^P$ for all $1 \leq i \leq n$. Then $\mathfrak{I}^{BV} \equiv_e^a \mathfrak{I}^P$ holds.

In the context of Algorithm 3, we further demand that the formulas returned by BVConstraints do not contain any propositional variables already occurring in C_{rcv}. In other words, all variables introduced by BVConstraints must be fresh. These requirements suffice to prove that the reduction to propositional logic performed by Algorithm 3 is correct.

In the following, we give a constructive definition of BVConstraints. Formally proving that it fulfills the before mentioned properties is often very technical, so we confine ourselves to an informal explanation of our construction, which should make it clear how to prove its correctness. The satisfiability under all variable assignments is guaranteed by enforcing that the formula set $\Phi = \text{BVConstraints}(e, b)$ only imposes restrictions on $e(b)$ (and on possibly introduced additional propositional variables, which do not appear in any other formula set $\Phi' = \text{BVConstraints}(e, b')$). In particular, the values of $V(e)$ that do not appear in $e(b)$ are not restricted by Φ. For any bitvector variable $v_{[n]}$, we require $\text{BVConstraints}(e, v) = \emptyset$. Together with the injectivity of e, these properties ensures that all created formula sets can be satisfied together for arbitrary variable assignments.

3.2.2 Encoding Simple Terms

We start our definition of $\text{BVConstraints}(e, b)$ with bitvector terms which are simple in the sense that they can be encoded without introducing new propositional variables.

The most basic bitvector expressions are expressions without subterms. In bitvector logic, all predicate symbols are of positive arity, such that the only expressions without subterms are terms of arity zero. Here we distinguish between interpreted terms (literals) and uninterpreted terms, i.e. bitvector variables.

Variables

For a bitvector variable $v_{[n]}$, we set $\text{BVConstraints}(e, v) := \emptyset$.

As previously explained, BVConstraints should not impose any restrictions on the values of $e(v)$, such that all possible assignments for v are considered.

Literals

Let c be a bitvector literal with the semantics $f_c = \lambda i \in \{\, 0, \ldots, n-1 \,\}.f_c(i)$ for some $n \in \mathbb{N}$. We encode the literal into the formula set:

$$\text{BVConstraints}(e, c) := \{\, e(c)_i \leftrightarrow f_c(i) \mid i \in \{\, 0, \ldots, n-1 \,\} \,\} \tag{3.1}$$

The remaining bitvector expressions b for which we have to define $\text{BVConstraints}(e, b)$ are all bitvector atoms and predicates of positive arity k. Let $b = \circ\mathrm{p}(t_1, \ldots, t_k)$ for a bitvector function or predicate symbol $\circ\mathrm{p}$. Then we set:

$$\text{BVConstraints}(e, b) := \Phi^{op}(e(t_1), \ldots, e(t_k), e(b)) \tag{3.2}$$

In the following, we define the functions Φ^{op} for all possible symbols *op*. These functions output sets of propositional constraints on the basis on the propositional variables (or, to be more general, the propositional terms) which are provided as arguments. We use the notation $a_{\langle n \rangle}$ to denote that a is a vector consisting of n terms, which are referenced by $a_{n-1}, \ldots, a_0$.

Bitwise Operators

The unary operator `bvnot` realizes a bitwise negation:

$$\Phi^{bvnot}(x_{\langle n\rangle}, t_{\langle n\rangle}) := \{\, t_i \leftrightarrow \neg x_i \mid i \in \{\, 0, \ldots, n-1 \,\} \,\} \tag{3.3}$$

The binary bitwise operators all share the semantics of a Boolean function that is applied element-wise, and their encoding only varies in the choice of the respective Boolean function. As an example, we present the encoding of `bvand`:

$$\Phi^{bvand}(x_{\langle n\rangle}, y_{\langle n\rangle}, t_{\langle n\rangle}) := \{\, t_i \leftrightarrow (x_i \wedge y_i) \mid i \in \{\, 0, \ldots, n-1 \,\} \,\} \tag{3.4}$$

An analogous construction is performed for the function symbols `bvand`, `bvnand`, `bvor`, `bvnor`, `bvxor` and `bxnor`.

The operator `bvcomp` performs a bitwise comparison and returns a bitvector with exactly one bit, which is 1 if and only if the compared bitvectors are equal. It is encoded by:

$$\Phi^{bvcomp}(x_{\langle n\rangle}, y_{\langle n\rangle}, t_{\langle 1\rangle}) := \left\{\, t_1 \leftrightarrow \bigwedge_{i=0}^{n-1} (x_i \leftrightarrow y_i) \,\right\} \tag{3.5}$$

Concatenation

Concatenation is implemented by relating each output bit to the respective bit in one of the two input vectors:

$$\begin{aligned} \Phi^{concat}(x_{\langle m\rangle}, y_{\langle n\rangle}, t_{\langle m+n\rangle}) := &\{\, t_i \leftrightarrow x_i \mid i \in \{\, 0, \ldots, m-1 \,\} \,\} \\ &\cup \{\, t_i \leftrightarrow y_{i-m} \mid i \in \{\, m, \ldots, n-1 \,\} \,\} \end{aligned} \tag{3.6}$$

The function $\texttt{repeat}_i$ could be rewritten into chains of `concat` applications, as in the definition, but it can also be encoded directly:

$$\Phi^{repeat_i}(x_{\langle n\rangle}, t_{\langle i\cdot n\rangle}) := \bigcup_{k=0}^{i-1} \{\, t_{k\cdot n+l} \leftrightarrow x_l \mid l \in \{\, 0, \ldots, n-1 \,\} \,\} \tag{3.7}$$

Extraction

The constraints for $\texttt{extract}_{i,j}$ are very similar to the ones for concatenation:

$$\Phi^{extract_{i,j}}(x_{\langle n\rangle}, t_{\langle j-i+1\rangle}) := \{\, t_k \leftrightarrow x_{j+k} \mid k \in \{\, 0, \ldots, i-j \,\} \,\} \tag{3.8}$$

Extension

For $i \in \mathbb{N}_0$, SMT-LIB defines two different extension operators, namely $\texttt{zero\_extend}_i$ and $\texttt{sign\_extend}_i$. Both operators share the semantics that they extend the width of their argument by i bits, while leaving its numerical representation unchanged. For `zero_extend`, an unsigned representation is taken as basis, such that an extension by i bits corresponds to a prepending of i bits of value 0. The `sign_extend` function can be realized in a similar way, due to a convenient property of the two's complement encoding: Instead of prepending bits with the constant value 0, the value of the most-significant bit is replicated. This covers both the positive and the negative case. Altogether, we obtain the following encodings:

$$\begin{aligned}\Phi^{zero\_extend_i}(x_{\langle n\rangle}, t_{\langle n+i\rangle}) := &\{\, t_k \leftrightarrow x_k \mid k \in \{\, 0, \ldots, n-1 \,\} \,\} \\ &\cup \{\, \neg t_k \mid k \in \{\, n, \ldots, n+i-1 \,\} \,\}\end{aligned} \tag{3.9}$$

$$\begin{aligned}\Phi^{sign\_extend_i}(x_{\langle n\rangle}, t_{\langle n+i\rangle}) := &\{\, t_k \leftrightarrow x_k \mid k \in \{\, 0, \ldots, n-1 \,\} \,\} \\ &\cup \{\, t_k \leftrightarrow x_{n-1} \mid k \in \{\, n, \ldots, n+i-1 \,\} \,\}\end{aligned} \tag{3.10}$$

Rotation

For defining the encoding of rotation operators, we use the function $\text{mod} : \mathbb{Z} \times \mathbb{N} \to \mathbb{N}_0$, given by $\text{mod}(a, n) = b \iff (a \equiv b \pmod{n}$ and $0 \leq b < m)$. Then the constraints for left and right rotation are almost identical:

$$\Phi^{rotate\_left_i}(x_{\langle n\rangle}, t_{\langle i\cdot n\rangle}) := \left\{\, t_k \leftrightarrow x_{\text{mod}(k-i,n)} \;\middle|\; k \in \{\, 0, \ldots, n-1 \,\} \,\right\} \tag{3.11}$$

$$\Phi^{rotate\_right_i}(x_{\langle n\rangle}, t_{\langle i\cdot n\rangle}) := \left\{\, t_k \leftrightarrow x_{\text{mod}(k+i,n)} \;\middle|\; k \in \{\, 0, \ldots, n-1 \,\} \,\right\} \tag{3.12}$$

3.2.3 Encoding Shifts

In a function predicate of the form $\texttt{bvshl}(i_{[n]}, d_{[n]})$ $(n \in \mathbb{N})$ we refer to $i_{[n]}$ as the shift input and call $d_{[n]}$ the shift distance (accordingly for `bvlshr` and `bvashr`).

As can be seen above, a rotation by i steps to the left or to the right can be encoded in a very straightforward way. Although shifting and rotation are two quite similar operations, encoding a shift operator from the SMT-LIB bitvector theory requires significantly more effort. This is due to the fact that the parameter i is not given as a fixed number, but as a bitvector term, which is not necessarily constant. Hence, different values for the shift distance need to be handled on the formula level.

We use a construction that is also employed in the hardware designs of most state-of-the-art CPUs, namely a *barrel shifter*. It works by splitting the shift operation into multiple stages, one for each bit of the shift distance. In each stage, either a shift by a fixed distance is performed or the result of the previous stage is left unchanged. The output of the last stage is the overall output of the operator.

Consider, for example, a shift distance $d_{[4]}$. In the first stage, the shift input is shifted by one bit if d_0 is 1. Otherwise, the output of the first stage is the unmodified shift input. In the second stage, the result of the previous stage is shifted by two bits if d_1 is 1, and left unchanged

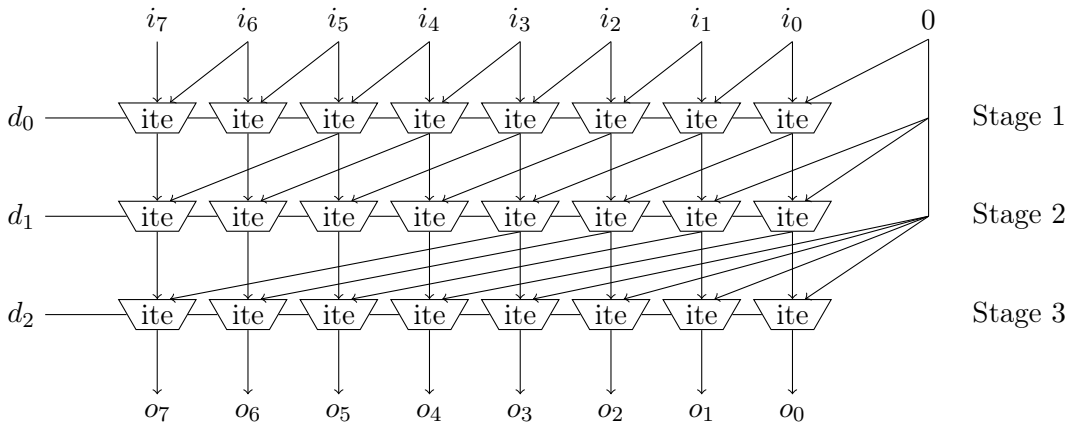


Figure 3.2: Barrel Shifter Network (left shift)

otherwise. The third stage conditionally shifts by 4 bits and the fourth stage by 8 bits. In general, stage i performs a shift by 2^{i-1} bits if d_{i-1} is 1. By proceeding like this, the shift with the previously unknown shift distance $d_{[4]}$ is replaced with 4 conditional shifts of fixed distance, which can easily be encoded similar to the rotation operators.

The SMT-LIB standard demands that the shift input and the shift distance are of an equal width n. This allows for an important observation: If the most significant bit of the shift distance is 1, the shift input has to be shifted by at least 2^{n-1} bits, which is always greater than or equal to n. As a result, the output of the shift operation is a bitvector only consisting of zeros. This effect typically applies to many more bits of the shift input. For $n = 16$, for example, any non-zero value among the 11 most-significant bits of the shift distance causes the result to be zero (since $2^{5-1} = 16 \geq n$).

To generalize the thought of the previous paragraph: For each bit position i of the shift distance with $2^i \geq n$, a value of 1 enforces an output value which is completely zero. We call this event "overshifting". Solving the equation for i, one can derive that overshifting occurs if any bit is 1 which has a position i with $i \geq \log_2(n)$. This observation can be used to optimize the encoding, as there is no need to encode full stages for bits that produce an overshift.

We now describe $\Phi^{bvshl}(x_{\langle n\rangle}, y_{\langle n\rangle}, t_{\langle n\rangle})$. Let $S := \lceil \log_2(n) \rceil$ be the number of stages that are generated. For each stage $s \in \{1, \ldots, S\}$, we compute the output of stage s in the fresh variables $x^s_{\langle n\rangle} = \langle x^s_{n-1}, \ldots, x^s_0 \rangle$. First, we define the following abbreviations:

- The constant *fill* describes as a propositional term the value that is inserted from the right during the left-shift. Since a left-shift always fills up with zeros, we can simply set:

$$\mathit{fill} := 0 \tag{3.13}$$

- The function stageBit(*stage*, *pos*) outputs a term for the bit value at position *pos* after the stage of the given number. We later use this function to refer to computation results from previous stages. If *pos* is outside of $\{0, \ldots, n-1\}$, the constant *fill* is returned. The input $x_{\langle n\rangle}$ is identified with a stage of number zero:

$$\text{stageBit}(\mathit{stage}, \mathit{pos}) := \begin{cases} \mathit{fill} & \text{if } \mathit{pos} \notin \{0, \ldots, n-1\} \\ x_{\mathit{pos}} & \text{if } \mathit{pos} \in \{0, \ldots, n-1\} \text{ and } \mathit{stage} = 0 \\ x^{\mathit{stage}}_{\mathit{pos}} & \text{if } \mathit{pos} \in \{0, \ldots, n-1\} \text{ and } \mathit{stage} > 0 \end{cases} \tag{3.14}$$

- Each stage s shifts conditionally by offset(s) positions, where:

$$\text{offset}(s) := 2^{s-1} \tag{3.15}$$

Using these abbreviations, the formula set $\Phi^{bvshl}(x_{\langle n\rangle}, y_{\langle n\rangle}, t_{\langle n\rangle})$ is defined as the set consisting of the following formulas:

- For $s \in \{1, \ldots, S\}, k \in \{0, \ldots, n-1\}$, the value of the variable x_k^s is determined by:

$$x_k^s \leftrightarrow (y_{s-1} \ ?\ \text{stageBit}(s-1, k-\text{offset}(s)) : \text{stageBit}(s-1, k)) \tag{3.16}$$

- The fresh variable $X_{overshift}$ is used to detect whether overshifting occurs:

$$X_{overshift} \leftrightarrow \bigvee_{k=S}^{n-1} y_k \tag{3.17}$$

- For $k \in \{0, \ldots, n-1\}$, the output bit t_k is either zero in the case of overshifting, or the corresponding bit of the last stage:

$$t_k \leftrightarrow (X_{overshift} \ ?\ \mathit{fill} : x_k^S) \tag{3.18}$$

A very similar encoding scheme is used for the `bvlshr` and `bvashr`. Shifting to the right instead of to the left is achieved by replacing the offset computation with its negative version $\text{offset}(\mathit{stage}) := -2^{\mathit{stage}-1}$. Furthermore, the arithmetic right-shift `bvashr` replicates the most-significant bit of the input $x_{\langle n\rangle}$ instead of filling up with zeros. This can be realized by setting $\mathit{fill} := x_{n-1}$.

3.2.4 Encoding Addition, Subtraction and Negation

Our construction of encoding formulas for the shift operators is strongly based on the logical circuits which can be found in modern CPUs. For the encoding of the addition operator `bvadd`, we proceed in a similar way and derive our formulas from well-known electronic addition networks.

The most important component in addition networks is called a *Full Adder*. It is capable of adding three binary values to a two-digit binary result. The least significant bit of the output is called the *sum bit*, the most significant bit is the *carry bit*. By modeling input and output bits as propositional variables, we can emulate the full adder as a set of formulas Φ^{FA}, which relates the input variables A_{in}, B_{in} and C_{in} to the sum bit S_{out} and the carry bit C_{out}:

$$\begin{aligned}\Phi^{FA}(A_{in}, B_{in}, C_{in}, S_{out}, C_{out}) := &\{\, S_{out} \leftrightarrow ((A_{in} \oplus B_{in}) \oplus C_{in}) \,\} \\ &\cup \{\, C_{out} \leftrightarrow ((A_{in} \wedge B_{in}) \vee ((A_{in} \oplus B_{in}) \wedge C_{in})) \,\}\end{aligned} \tag{3.19}$$

The first formula in the set states that the sum bit S_{out} should be 1 if and only if exactly one or three of the input variables are of value 1 (which corresponds to the sum `01` or `11`). The carry bit C_{out} should be 1 if and only if at least two of the input variables are 1 (giving the sum `10` or `11`).

Building on the full adder component, one can construct networks which add numbers consisting of multiple digits. For adding two n-digit numbers, a sequence of n full adders is created,

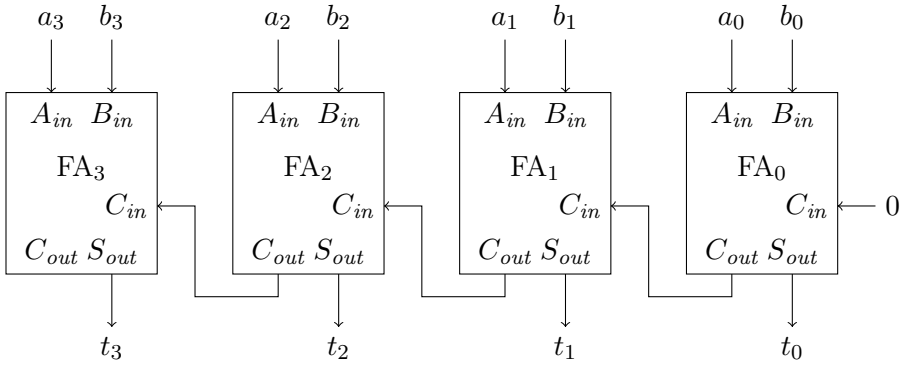


Figure 3.3: Adder network

which are chained as illustrated in Figure 3.3. Here the meaning of the term "carry bit" becomes obvious: The carry bit C_{out} produced by the full adder with number i is "carried" to the full adder with number $i+1$, where it appears as the third input bit C_{in}. As a circuit, this composition is called a *ripple-carry adder*.

We can now construct $\Phi^{bvadd}(x_{\langle n \rangle}, y_{\langle n \rangle}, t_{\langle n \rangle})$ by translating the ripple-carry adder into propositional constraints as follows: For $i \in \{ 0, \ldots, n \}$, let C_i be a fresh variable. These introduced variables are used to represent the carry bits in the summation of x and y. We can then set:

$$\Phi^{bvadd}(x_{\langle n \rangle}, y_{\langle n \rangle}, t_{\langle n \rangle}) := \{ \neg C_0 \} \cup \bigcup_{i=0}^{n-1} \Phi^{FA}(x_i, y_i, C_i, t_i, C_{i+1}) \tag{3.20}$$

Note that the calculation for the least significant output bit differs from the remaining digits: For bit position 0, no carry-in from previous digits exists, such that the summation for this bit is only a sum over two instead of three bits. Consequently, one could replace the encoding of a full adder by the encoding of an adder which only takes two input bits. Such an adder is typically referred to as a *half adder*. In our implementation, we stick to the full adder instead and force its third input bit (C_0) to be 0.

It should be remarked that the most-significant carry bit C_n has a special role in the addition of unsigned integers. If it is 1, we know that the sum of a and b cannot be represented using n bits, such that an overflow has occurred in the calculation. A similar observation can be made for signed integer addition. This idea of overflow detection becomes relevant in Section 3.2.5. At this point, however, we do not need any detection or correction mechanisms, as the overflow behavior is already reflected in the semantics of `bvadd`.

We now present how the negation operator `bvneg` can be encoded into SAT. Here we make use of an important property of the two's complement representation:

Lemma 3.2.1. *Let $n \in \mathbb{N}$. For any bitvector $b \in BitVec_n$ the following equality holds:*

$$\text{bv2nat}_n(\text{bvnot}(b)) = 2^n - \text{bv2nat}_n(b) - 1 \tag{3.21}$$

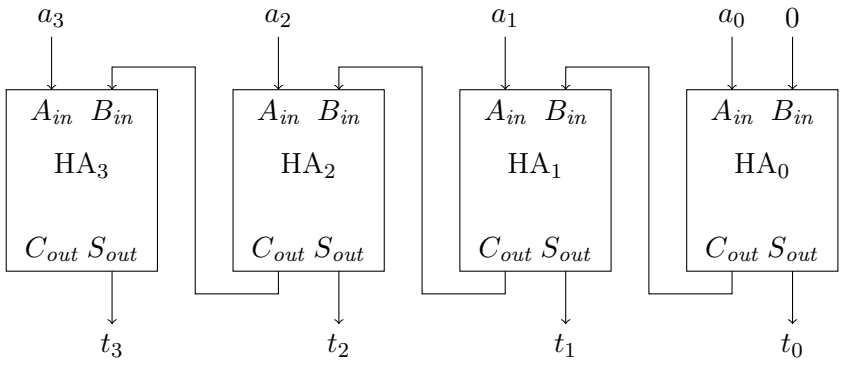


Figure 3.4: Incrementing network

Proof. We have:

$$\begin{aligned}
\text{bv2nat}_n(\text{bvnot}(b)) &= \sum_{i=0}^{n-1} (1 - b_i) \cdot 2^i \\
&= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} b_i \\
&= \sum_{i=0}^{n-1} (2^{i+1} - 2^i) - \text{bv2nat}_n(b) \\
&= 2^n - 2^0 - \text{bv2nat}_n(b) \\
&= 2^n - \text{bv2nat}_n(b) - 1
\end{aligned}$$

□

As the semantics of `bvneg` are defined modulo 2^n, we can conclude from Lemma 3.2.1 that we can rewrite $\texttt{bvneg}(b_{[n]})$ as:

$$\text{bvneg}(b) = \text{bvadd}(\text{bvnot}(b), \text{nat2bv}_n(1)) \tag{3.22}$$

Hence, we can negate a bitvector $b_{[n]}$ by flipping each bit and adding 1. In principle, this can be achieved by applying the construction for `bvadd` to the arguments $x := \text{bvnot}(b)$ and $y := \text{nat2bv}_n(1)$. However, it strikes that in each step of the addition, only two bits need to be added, as the third one is always zero. Here we can obtain much easier formulas by changing over to the herein before mentioned half adders, which have the same semantics as full adders, except that they only add up two instead of three bits:

$$\begin{aligned}
\Phi^{HA}(A_{in}, B_{in}, S_{out}, C_{out}) := &\{\, S_{out} \leftrightarrow (A_{in} \oplus B_{in}) \,\} \\
&\cup \{\, C_{out} \leftrightarrow (A_{in} \wedge B_{in}) \,\}
\end{aligned} \tag{3.23}$$

Incrementing an n-bit bitvector value by 1 can be achieved by arranging n half adders in the scheme depicted in Figure 3.4. Conclusively, we can encode a term $t = \texttt{bvneg}(x_{[n]})$ by incrementing the complement of x in the following manner:

$$\Phi^{bvneg}(x_{\langle n \rangle}, t_{\langle n \rangle}) := \{\, C_0 \,\} \cup \bigcup_{i=0}^{n-1} \Phi^{HA}(\neg x_i, C_i, t_i, C_{i+1}) \tag{3.24}$$

With these components, a subtraction $t = \texttt{bvsub}(x_{[n]}, y_{[n]})$ can easily be implemented by rewriting it to $\texttt{bvadd}(x, \texttt{bvneg}(y))$. An optimized variant is not to compute $\texttt{bvneg}(y)$ explicitly, but to combine the formulas for addition and negation directly. This leads to the implementation:

$$\Phi^{bvsub}(x_{\langle n\rangle}, y_{\langle n\rangle}, t_{\langle n\rangle}) := \{\, C_0 \,\} \cup \bigcup_{i=0}^{n-1} \Phi^{FA}(x_i, \neg y_i, C_i, t_i, C_{i+1}) \tag{3.25}$$

3.2.5 Encoding Relational Operators

The most fundamental relational operator, which is already part of the SMT-LIB Core theory, is the equality operator. It is translated into a conjunction of bitwise comparisons:

$$\Phi^{=}(a_{\langle n\rangle}, b_{\langle n\rangle}, E) := \{\, E \leftrightarrow \bigwedge_{i=0}^{n-1} (a_i \leftrightarrow b_i) \,\} \tag{3.26}$$

For the comparison operator `bvult`, we can proceed as follows. Given two bitvectors $a_{[n]}$ and $b_{[n]}$, the function value $\text{bvult}(a, b)$ is 1 if and only if $\text{bv2nat}(a) < \text{bv2nat}(b)$, which is equivalent to the condition $\text{bv2nat}(a) - \text{bv2nat}(b) < 0$.

As presented in the construction of Φ^{bvsub} in Section 3.2.4, a subtraction modulo 2^n is realized by applying an addition network to the arguments a and $\text{bvnot}(b)$ with a carry-in bit of 1. We can then use the carry-out bit C_n of the network to detect whether an overflow occurred in the addition, which reveals the relation of a and b:

$$\begin{aligned} C_n \text{ is } 0 &\iff \text{bv2nat}(a) + \text{bv2nat}(\text{bvnot}(b)) + 1 < 2^n \\ &\iff \text{bv2nat}(a) + (2^n - \text{bv2nat}(b) - 1) + 1 < 2^n \\ &\iff \text{bv2nat}(a) - \text{bv2nat}(b) < 0 \\ &\iff \text{bv2nat}(a) < \text{bv2nat}(b) \end{aligned} \tag{3.27}$$

Hence, we can implement the comparison operator bvult by the following formula set (where $s_{\langle n\rangle}$ is a vector of n fresh variables):

$$\Phi^{bvult}(a_{\langle n\rangle}, b_{\langle n\rangle}, R) := \Phi^{bvsub}(a, b, s) \cup \{\, R \leftrightarrow \neg C_n \,\} \tag{3.28}$$

For the signed variant of the "$<$" comparison operator, `bvslt`, we distinguish between two cases. If $a_{[n]}$ and $b_{[n]}$ have the same sign (i.e. the values of their most-significant bits are identical), we can use the output of `bvult` because $\text{bvslt}(a, b) = \text{bvult}(a, b)$ holds. Otherwise the output of `bvult` is inverted (because signed negative numbers start with 1 and are thereby greater than signed non-negative numbers, when being interpreted as unsigned values):

$$\Phi^{bvslt}(a_{\langle n\rangle}, b_{\langle n\rangle}, R) := \Phi^{bvult}(a, b, U) \cup \{\, R \leftrightarrow (U \leftrightarrow (a_{n-1} \leftrightarrow b_{n-1})) \,\} \tag{3.29}$$

The remaining relational operators `bvule`, `bvugt`, `bvuge`, `bvsle`, `bvsgt` and `bvsge` can easily be reduced to Φ^{bvult} or Φ^{bvslt} by reversing the order of the arguments, negating the output or both.

A_2	A_1	A_0	$\cdot$	B_2	B_1	B_0
				$A_2 \wedge B_0$	$A_1 \wedge B_0$	$A_0 \wedge B_0$
	$+$		$A_2 \wedge B_1$	$A_1 \wedge B_1$	$A_0 \wedge B_1$	0
	$+$	$A_2 \wedge B_2$	$A_1 \wedge B_2$	$A_0 \wedge B_2$	0	0
				S_2	S_1	S_0

Figure 3.5: Shift and Add: Multiplying two three-digit numbers $A_2A_1A_0$ and $B_2B_1B_0$ by adding partial products

3.2.6 Encoding Multiplication, Division and Remainder

As for shifts and additions, we also regard the implementation of multiplication on the hardware level as a prototype for an encoding to SAT. In the case of multiplication, different approaches exist and several optimizations have been proposed and implemented, with particular focus on how to arrange the logical network in a way that it has good support for parallel evaluation. These suggestions lead to a better performance, but usually also involve more logical gates and more complexity, which make it doubtful whether an encoding to SAT would benefit from such a design.

In our construction, we realize multiplication by a classical approach which resembles long multiplication, e.g. in the decimal system. In order to multiply two numbers a and b, a partial product is calculated for each digit of b. All partial products are then added up to the final result. Applying this method to the binary system, one obtains an algorithm named *shift and add*. Its name stems from the performed operations: In the binary system, each partial product is either 0 or the bit sequence a, which is shifted to the left. Adding all products then produces the desired output.

Figure 3.5 shows the scheme of "shift and add", which we now translate into propositional logic for two input vectors $x_{\langle n \rangle}, y_{\langle n \rangle}$ and the output vector $t_{\langle n \rangle}$.

For each line $l \in \{ 0, \ldots, n-1 \}$ of the computation, we introduce an n-elementary vector of fresh variables $p^l_{\langle n \rangle} := \langle p^l_{n-1}, \ldots, p^l_0 \rangle$ to represent the product calculated in this line. The multiplication with 0 or 1 in each cell is realized using a logical And. Note that the output vector t is also of length n, such that we only need to consider the n least-significant bits of each line. For $l, i \in \{ 0, \ldots, n-1 \}$, this gives the formula:

$$p^l_i \leftrightarrow \begin{cases} x_{i-l} \wedge y_l & \text{if } i \geq l \\ 0 & \text{if } i < l \end{cases} \tag{3.30}$$

Starting with line number 1 (i.e., the second line), we sum up all previous intermediate results in the vector $s^l_{\langle n \rangle} := \langle s^l_{n-1}, \ldots, s^l_0 \rangle$ of fresh variables, using our formulas for an addition network. For $l \in \{ 1, \ldots, n-1 \}$, we add the formulas:

$$\Phi^{bvadd}(s^{l-1}, p^l, s^l) \tag{3.31}$$

(Note that each call to Φ^{bvadd} creates fresh variables for the carry bits. In other words, the variable C_i in the set $\Phi^{bvadd}(s^2, p^1, s^1)$ is different from C_i appearing in $\Phi^{bvadd}(s^1, p^0, s^0)$. This is ensured by renaming the carry variables appropriately.)

Finally, the result t is set to the last generated sum s^{n-1}. For $i \in \{0, \ldots, n-1\}$, we add (not regarding the special case $n = 1$, which is trivial to implement anyway):

$$t_i \leftrightarrow s_i^{n-1} \tag{3.32}$$

The set $\Phi^{bvmul}(x_{\langle n \rangle}, y_{\langle n \rangle}, t_{\langle n \rangle})$ is defined as the union of the three previously described formula sets.

The fact that we use $n-1$ adders for the encoding of t make multiplication a very costly operator when translated to SAT. Here the overhead of a propositional encoding of arithmetic operations becomes apparent, which is still one of the biggest downsides of this approach.

The last remaining arithmetic operations are division and remainder. Typically all implementations of division in modern CPUs make use of some iterative algorithm, which build approximations of the quotient and successively improve them until some criteria is met. Programmatically, this corresponds to a loop with a certain termination condition. Imitating a loop on the SAT level unfortunately is not straightforward at all. An unrolling of the loop is not possible as the number of iterations is usually not known upfront. Apart from that, many division algorithms make intense use of repetitive multiplication, which we would like to avoid for performance reasons.

Instead, let us recall the definition of division and remainder: For a non-negative integer a and a positive integer b, we call $q \in \mathbb{N}_0$ the quotient and $r \in \mathbb{N}_0$ the remainder of the division of a and b if the following relationship holds:

$$a = q \cdot b + r \text{ with } 0 \leq r < b \tag{3.33}$$

This gives rise to the following idea of encoding division and remainder: Instead of imitating the calculation of q and r by formulas, we encode the above constraint and leave the actual computation to the SAT solver. Thereby we obtain the following constraints for the operators `bvudiv` and `bvurem`:

$$\Psi^{bvudiv}(a_{\langle n \rangle}, b_{\langle n \rangle}, q_{\langle n \rangle}) := \Phi^{\overline{bvmul}}(q, b, p') \ \cup \ \Phi^{\overline{bvadd}}(p', r', a) \ \cup \ \Phi^{ult}(r', b) \tag{3.34}$$

$$\Psi^{bvurem}(a_{\langle n \rangle}, b_{\langle n \rangle}, r_{\langle n \rangle}) := \Phi^{\overline{bvmul}}(q', b, p') \ \cup \ \Phi^{\overline{bvadd}}(p', r, a) \ \cup \ \Phi^{ult}(r, b) \tag{3.35}$$

Here p', q' and r' are vectors of n fresh variables. The adder and multiplier formula sets $\Phi^{\overline{bvadd}}$ and $\Phi^{\overline{bvmul}}$ are very similar to the previously presented Φ^{bvadd} and Φ^{bvmul}, except that they do not permit overflows in the calculation, which is crucial for the implementation (modular semantics of the operators $\cdot$ and $+$ in Equation 3.33 break the definition). The non-overflowing operator $\Phi^{\overline{bvadd}}$ is obtained by adding the additional constraint that the carry-out for the most-significant digit must be zero:

$$\Phi^{\overline{bvadd}}(a_{\langle n \rangle}, b_{\langle n \rangle}, t_{\langle n \rangle}) := \Phi^{bvadd}(a, b, t) \cup \{ \neg C_n \} \tag{3.36}$$

Based on this, one can obtain the non-overflowing variant $\Phi^{\overline{bvmul}}$ from Φ^{bvmul} by replacing all occurrences of Φ^{bvadd} with $\Phi^{\overline{bvadd}}$. Additional constraints are added to ensure that all expressions on the left of the dashed line in Figure 3.5 are zero. It should be noted that the non-overflowing

extensions no longer encode total, but only partial functions, which are not satisfiable for interpretations under which the addition or multiplication of the inputs a and b would cause an overflow. For our construction of the division and remainder constraints, this is exactly the desired behavior.

Special care has to be taken to ensure the correct modeling of division by zero. The SMT-LIB standard states that no assumptions about division function values like $\text{bvudiv}(x_{[n]}, y_{[n]})$ should be permitted in the case that y only consists of zeros. To put it differently, expressions of this kind may be interpreted by any n-bit bitvector value. We model this special case by not restricting the output variables if the value of y is zero:

$$\Phi^{bvudiv}(a_{\langle n\rangle}, b_{\langle n\rangle}, q_{\langle n\rangle}) := (\bigvee_{i=0}^{n-1} b_i) \rightarrow \bigwedge \Psi^{bvudiv}(a, b, q) \tag{3.37}$$

$$\Phi^{bvurem}(a_{\langle n\rangle}, b_{\langle n\rangle}, r_{\langle n\rangle}) := (\bigvee_{i=0}^{n-1} b_i) \rightarrow \bigwedge \Psi^{bvudiv}(a, b, r) \tag{3.38}$$

The signed operators `bvsdiv`, `bvurem`, `bvsrem` and `bvsmod` are defined in the SMT-LIB standard as case distinctions which ultimately fall back to their unsigned counterparts. Therefore, they can easily be implemented as if-then-else-expressions using `bvudiv` and `bvurem`, which is why we omit a detailed presentation here.

3.3 Optimizations

In this section we present a number of optimizations which we implemented for an improved efficiency of our solver.

Structural Hashing

The decision algorithm which we implemented for bitvector arithmetic is based on the notion of a bitvector mapping e (see Definition 3.1.1 for details), which maps bitvector expressions to propositional variables. The definition implies that two occurrences of the same subterm in a bitvector formula are always mapped to the same vector of Boolean variables. From a technical point of view, this makes it important to have a quick way of deciding whether a given term t already has a function value $e(t)$ assigned or not. Throughout the remaining algorithm, we also make heavy use of e in the construction of propositional constraints by the function BVConstraints. Again, it is crucial that we have a fast way of looking up a function value $e(t)$ for a given term t.

For this purpose, the bitvector mapping is implemented as a `std::unordered_set`, which is a key-value store that supports a lookup with a constant average time complexity. Internally, this data structure organizes its elements via a hash function over the key datatype. We implemented an efficient hash function for our data structures for bitvector terms and atoms, and compute the hash value for every instance during its initialization. This concept is called *structural hashing*. As an additional optimization, mainly in terms of memory allocation, we manage all bitvector terms in a dedicated manager class, which ensures that identical terms are only represented once in memory. Again, this functionality is based on structural hashing.

Non-injective Bitvector Mappings

In Section 3.2.1 the semantics of the BVConstraints method are defined on the basis of an injective bitvector mapping. The property of injectivity ensures that each bit of a bitvector expression is represented by a dedicated propositional variable, such that no encoding conflicts can arise due to the choice of the bitvector mapping. However, with the intention to reduce the number of required propositional variables, it may be desirable to weaken this constraint. If it is clear that two variables always have the same truth value assigned under any satisfying interpretation, we permit that this pair of variables is represented by a single, shared variable instead.

As an example, consider the term $t_{[3]} = \texttt{extract}_{2,0}(b_{[6]})$. In an injective bitvector mapping e, we might use the variables $e(b) = \langle B_5, B_4, B_3, B_2, B_1, B_0 \rangle$ and $e(t) = \langle T_2, T_1, T_0 \rangle$. The call to BVConstraints(e, t) would then generate the constraints $\{ T_2 \leftrightarrow B_2, T_1 \leftrightarrow B_1, T_0 \leftrightarrow B_0 \}$. As one can see, there is no real need for own propositional variables in $e(t)$, since they are constrained to be equivalent to the lower three variables of $e(b)$. Instead, we might as well set $e(t) = \langle B_2, B_1, B_0 \rangle$ and BVConstraints$(e, t) := \emptyset$, which leads to an equisatisfiable and more compact resulting SAT instance.

To obtain such optimized encodings, we modified Algorithm 3 in a way that it does not generate the bitvector mapping upfront, but instead allows the function BVConstraints to extend the (initially empty) bitvector mapping ad-hoc. Before BVConstraints is called on a term t, it is called for all subterms $t_0, \ldots, t_k$ of t, such that $e(t_0), \ldots, e(t_k)$ are already defined. The function call BVConstraints(e, t) may then compose $e(t)$ from fresh variables and those from $e(t_0), \ldots, e(t_k)$.

Bitvector operators which highly benefit from this concept are `concat`, `extract`, `repeat`, `zero_extend`, `sign_extend`, `rotate_left` and `rotate_right`. For most of them, BVConstraints can return an empty set, as the semantics are already realized by an appropriate choice of the bitvector mapping.

Mapping to Terms Instead of Variables

A further relaxation of the bitvector mapping definition can lead to massively optimized formulas in relation to constant or partially constant bitvector expressions. As an example, let $t_4 = \texttt{bvor}(b_{[4]}, \texttt{0b0110})$. So far, the bitvector literal $l = \texttt{0b0110}$ is encoded by a set of variables $e(l) = \langle L_3, L_2, L_1, L_0 \rangle$ with the constraints $\{ \neg L_3, L_2, L_1, \neg L_0 \}$. The concept of the bitvector mapping requires a 4-elementary bitvector to represent l, but it seems unnecessary to introduce variables whose values are restricted to be always 0 or always 1. In the spirit of the previously mentioned non-injective bitvector mappings, we might instead encode into $e(l) = \langle L_f, L_t, L_t, L_f \rangle$ with the constraints $\{ \neg L_f, L_t \}$, and thereby encode all true values with the same variable L_t and all false values with L_f. This saves two variables, but still does not seem to be the optimal solution.

Instead of forcing $e(t)$ to be a vector of variables, we now allow $e(t)$ to be a vector of propositional terms, each of which either consists of a single variable or the constant 0 or 1. Thereby, we can encode l into $e(l) = \langle 0, 1, 1, 0 \rangle$ without any further constraints. The main benefit becomes obvious when encoding t: Assume that $e(b) = \langle B_3, B_2, B_1, B_0 \rangle$. Previously, we would have encoded t into $e(t) = \langle T_3, T_2, T_1, T_0 \rangle$ with the constraints:

$$\{ T_3 \leftrightarrow (B_3 \vee L_3), T_2 \leftrightarrow (B_2 \vee L_2), \ldots \} \tag{3.39}$$

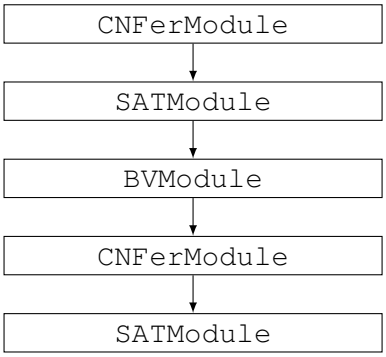


Figure 3.6: A strategy graph for full lazy flattening of SMT(QF_BV)

Now, with the improved encoding $e(l)$, we obtain the constraints:

$$\{ T_3 \leftrightarrow (B_3 \vee 0), T_2 \leftrightarrow (B_2 \vee 1), \dots \} \tag{3.40}$$

By applying simplification rules to these constraints, it becomes apparent that we can encode the whole term t by $e(t) = \langle B_3, 1, 1, B_0 \rangle$ without any constraints at all. Our relaxation of the bitvector mapping makes it possible to propagate the constant values appearing in l to the encoding of t.

This optimization applies to all bitvector terms t for which the output values $e(t)$ have previously been set via BVConstraints(e, t) using constraints of the form $e(t)_i \leftrightarrow t_i$ for some propositional terms t_i. We now proceed as follows: First, we try to simplify t_i by eliminating all contained constants. If the result t_i' is a single variable or a constant 0 or 1, we set $e(t)_i := t_i'$. Otherwise, we create a fresh variable for $e(t)_i$ as done before, and create the constraint $e(t)_i \leftrightarrow t_i'$.

Depending on the concrete example, the number of created variables from bit-blasting can be significantly reduced by the beforementioned technique, leading to measurable performance gains.

Full Lazy Flattening

At the beginning of this chapter, we showed in Figure 3.1 a possible embedding of our `BVModule` in a strategy graph for SMT-RAT. In the illustrated composition, we call the realized solving strategy *eager flattening*, as explained in Section 2.3.2, because bitvector logic is eliminated first before calling the SAT solving engine.

As a result of the configuration flexibility of SMT-RAT, we can easily switch to full lazy flattening instead, by choosing a strategy as depicted in Figure 3.6. Here, we realize a composition in the style of DPLL(T), where the outer SAT solver assigns truth values to the bitvector constraints, which are then passed to the `BVModule`. Due to its reduction to SAT, the `CNFerModule` and the `BVModule` appear a second time in the strategy graph to serve as a backend of the `BVModule`.

A great evaluation of lazy and eager techniques for bitvector arithmetic is given in [HBJ+14].

Chapter 4

Solving Integer Arithmetic

In the previous chapter, we described the implementation of a solver for bitvector arithmetic which is based on bit-blasting, i.e. the translation of constraints into propositional logic. Although the method of bit-blasting is most frequently encountered in combination with bitvector logic, the approach can lead to good results for other logics as well. This chapter presents an application of bit-blasting to linear and nonlinear integer arithmetic. It is based on a reduction from SMT(QF_NIA) to SMT(QF_BV), such that we can reuse the functionality provided by the previously explained bitvector module.

The following Section 4.1 gives an overview of the basic concept and the composition of our SMT-RAT module for integer arithmetic. Here, we omit technical details in favor of a high-level presentation of our decision algorithm. Sections 4.2 and 4.3 fill the remaining gaps by elaborating on various aspects of the implementation in depth. In Section 4.4 we describe a major extension of our algorithm, which employs a technique named Interval Constraint Propagation to make the reduction more efficient, and, at the same time, to decide satisfiability for some inputs completely without the need for a reduction. We use Section 4.5 to discuss some important aspects of making our module SMT-compliant. Again, we conclude the chapter by outlining several minor optimizations in Section 4.6.

4.1 IntBlastModule Overview

In Section 2.4.2 we discussed the most important decision procedures which are currently used for integer arithmetic. The majority of the presented approaches can be classified as some kind of adaptation of a decision procedure for real arithmetic. Typically, these adaptations, which add the restriction of integrality, do not come in naturally in the sense that they would preserve the good algorithmic properties of the original procedures. On the one hand, this is not completely surprising, as we already stated that the satisfiability problem is inherently more difficult over the integers than over the reals. On the other hand, it remains the question whether methods for a continuous search space can really serve as a good basis for operating on a discrete domain.

Following these considerations, we opted for a SAT-driven approach, which basically employs similar techniques as AProVE [FGM+07] and Z3 [DMB08]. Given a set of Diophantine constraints, we first restrict the ranges of all integer variables to a configurable cardinality. Each integer variable is encoded into a vector of propositional variables using its binary representation. After replacing all arithmetic operations with appropriate propositional constraints, the resulting formulas are handed over to a SAT solver. If satisfiability is detected, the input is satisfiable as

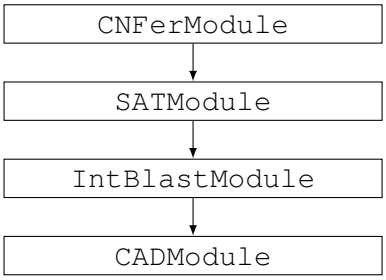


Figure 4.1: A strategy graph for SMT(QF_NIA)

well; if unsatisfiability is reported, other decision procedures need to be consulted.

In contrast to the two mentioned other implementations, we do not directly encode from integer arithmetic to SAT, but instead use bitvector arithmetic as an intermediate step. Proceeding like this, the concerns are clearly separated between our two modules: The `IntBlastModule`, which is illustrated in this chapter, is responsible for restricting the variable ranges, deciding about how to represent them as bitvectors, creating suitable representations of intermediate terms and finally encoding into bitvector logic. Afterwards, the `BVModule` maps the introduced bitvectors to propositional variables and applies its propositional encodings of arithmetic operators.

By choosing this two-step method, we do not only avoid redundancy in our implementation, but also obtain a better modularity. Bit-blasting is supported by all state-of-the-art solvers for bitvector logic, such that we could combine our `IntBlastModule` with any external bitvector solver. Moreover, the bitvector language already has built-in support for negative numbers, which we can easily make use of.

Our `IntBlastModule` is built as a pure theory module in the concept of DPLL(T). More specifically, it expects its input formulas to be inequalities over the integers, without any symbols from different theories and without any Boolean connectives. As such, its position in a strategy graph should be behind the SAT solving module, as illustrated in Figure 4.1. The backend of the `IntBlastModule` (in the example, the `CADModule`) serves as a fallback for the case that the bitvector reduction is not satisfiable. As this only implies that the input formulas are unsatisfiable over the restricted variable ranges, the backend proceeds by searching for solutions outside of these ranges.

Internally, the `IntBlastModule` uses a solver for bitvector arithmetic, which we refer to as `BVSolver`. The strategy of the solver is again specified by stating its strategy graph, for which we use the configuration from Figure 3.1. Unfortunately, the current design of SMT-RAT does not support multiple backends receiving different formula sets, which is why we had to put the `BVSolver` inside the `IntBlastModule` instead of embedding it into the global strategy graph. A graphical illustration of our module can be found in Figure 4.2. For the sake of completeness, the diagram also shows an `ICPModule` as an inner component, whose purpose is explained in Section 4.4. Until then, we will ignore this module in our further presentation, since it only provides an optional optimization.

Algorithm 4 shows the decision procedure implemented in our `IntBlastModule`. To give an understanding of the whole algorithm, we now explain its principles without going too much into the details of all utilized functions. For a more in-depth discussion, we refer to the following Sections 4.2 and 4.3.

At the beginning of the decision algorithm, each of the input constraints is preprocessed by

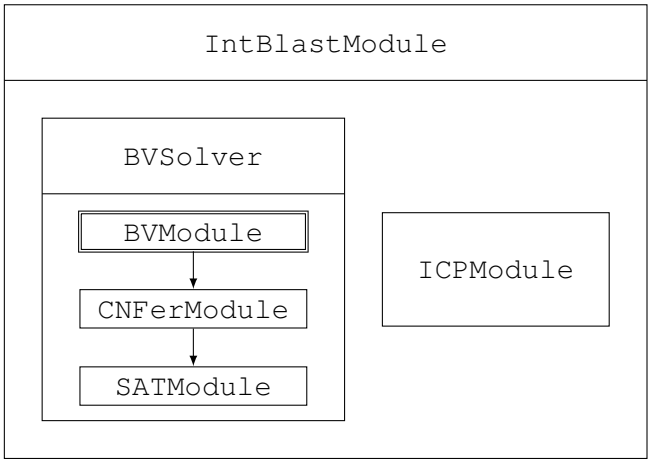


Figure 4.2: Internal composition of the `IntBlastModule`

```
1: function IntBlastModule.decide(input formulas C_rcv)
2:     C ← { rewrite(c) | c ∈ C_rcv }
3:     e ← createFreshIntegerMapping(C)
4:     C_BV ← ∅
5:     for all c ∈ C do
6:         C_BV ← C_BV ∪ encode(e, c)
7:     end for
8:     if BVSolver.decide(C_BV) = sat then
9:         return sat
10:    else
11:        C_pas ← C_rcv ∪ { ¬ inBounds(e) }
12:        return backend.decide(C_pas)
13:    end if
14: end function
```

Algorithm 4: IntBlastModule.decide

the function REWRITE. It first brings every input formula into an equivalent canonical form by expanding products, merging terms with the same monomials and rearranging the summands according to a fixed ordering. Then the inequality is transformed into a tree structure which fixes an evaluation order of the constraint. Most importantly, the tree structure defines how associative operators should be evaluated. As an example, the term $2xy$ can be evaluated as $2 \cdot (x \cdot y)$ or as $(2 \cdot x) \cdot y$, which corresponds to different evaluation trees. The formal details are explained in Section 4.2.

After the evaluation order has been fixed, the function CREATEFRESHINTEGERMAPPING makes a decision on how each integer variable should be represented as a bitvector. Its behavior is comparable to the function CREATEBITVECTORMAPPING in Algorithm 3. Given an integer variable x, the function generates a new bitvector variable $b_{[n]} \in \mathit{BitVec}_n$ for a suitable width n. Additionally, an annotation a for the new bitvector is generated, which states, for example, whether the value in $b_{[n]}$ should be interpreted as a signed or an unsigned value. Together, the tuple $\langle b_{[n]}, a \rangle$ is referred to as an *annotated bitvector variable*, and the return value of CREATEFRESHINTEGERMAPPING is a mapping from each integer variable occurring in C to its respective annotated bitvector variable. In analogy to the previous chapter, we call this mapping an *integer mapping* (a more precise specification follows in Definition 4.3.5).

It is important to note that the choice of the integer mapping e implies for each integer variable x an interval $[\min_e(x), \max_e(x)]$ that is representable on the basis of e. The further reduction to bitvector arithmetic is made under the assumption that $\min_e(x) \leq x \leq \max_e(x)$ for each integer variable x. We make this condition explicit in the following formula:

$$\text{inBounds}(e) := \bigwedge_{x \in \text{dom}(e)} (\min_e(x) \leq x \wedge x \leq \max_e(x)) \tag{4.1}$$

The formula inBounds(e) expresses the border which divides the search space into a part that is searched by the bitvector solver, and a part that is searched by the backend module. More formally, the `IntBlastModule` constructs the sets C_{BV} and C_{pas} such that C_{BV} is equisatisfiable to $C_{rcv} \cup \{ \text{inBounds}(e) \}$ and C_{pas} is equisatisfiable to $C_{rcv} \cup \{ \neg \text{inBounds}(e) \}$.

In lines 4 to 7, the formula set C_{BV} is composed by calling the ENCODE function for every rewritten input constraint. Every subterm of the arithmetic constraint is rewritten into a bitvector term, substituting addition and multiplication by calls to `bvadd` and `bvmul`. Integer variables are replaced by bitvector variables, as defined by the integer mapping e. Special care has to be taken to ensure that all terms for intermediate results are encoded into bitvectors of sufficient width, such that no overflow effects occur. Section 4.3 provides more information about how the ENCODE function works and how the involved term widths are calculated.

Finally, the `IntBlastModule` calls its internal bitvector solver on the set C_{BV} in Line 8. If the output is *sat*, we know that the set C_{rcv} is satisfiable (even with the additional constraint inBounds(e)). Otherwise, $C_{rcv} \models \neg \text{inBounds}(e)$ holds and the backend is called on the remaining search space.

The following two sections give a more fine-grained explanation of some crucial parts of the algorithm.

4.2 Constraint Rewriting

Linear and nonlinear arithmetic constraints in SMT-RAT are represented as multivariate polynomials which are compared to zero. Thus, every constraint is of the form $p \sim 0$ where p is a polynomial and $\sim \in \{ =, \neq, <, \leq, >, \geq \}$. The polynomial p is stored fully expanded as a sum of terms, where each term consists of a non-zero coefficient and a monomial (which may be empty). It is ensured that no two terms with the same monomials exist. All arithmetic constraints from the input are transformed into such a representation during the parsing process.

Expressing constraints in the described manner has several benefits. Two constraints can easily be compared by the terms in their polynomials, and arithmetic solving procedures are able to simplify sets of constraints with the same relational symbol $\sim$ by adding or subtracting the involved polynomials. However, when attempting to translate a polynomial into a bitvector term using the operators `bvadd` and `bvmul`, two challenges arise:

1. Arithmetic addition and multiplication are associative, such that we do not differentiate between the polynomials $(a \cdot b) \cdot c$ and $a \cdot (b \cdot c)$. When encoding into the operators `bvadd` and `bvmul`, on the other hand, we need to decide whether to encode into $\texttt{bvadd}(\texttt{bvadd}(a', b'), c')$ or the syntactically different term $\texttt{bvadd}(a', \texttt{bvadd}(b', c'))$.
2. Similarly, arithmetic addition and multiplication are commutative. It is possible to encode the sum $a + b$ into either $\texttt{bvadd}(a', b')$ or $\texttt{bvadd}(b', a')$.

In both cases, the encoding process needs to make a choice about the evaluation order. While this choice has no effect on the satisfiability of the encoded formula, it does lead to different SAT encodings and may thereby influence the performance of the bitvector solver. In general, it is desirable to minimize the number of distinct `bvadd` and `bvmul` applications among the encodings of a set of polynomials, such that the bitvector solver only generates as few costly multiplication and addition networks in total as possible.

The purpose of the REWRITE function in Algorithm 4 is to make a decision on the evaluation order for each arithmetic constraint. Throughout the remaining algorithm, this evaluation order remains fixed and is applied in the actual encoding process inside the ENCODE function. More specifically, the REWRITE function transforms each constraint into a tree structure, which incorporates the choices on the evaluation order.

Definition 4.2.1 (Polynomial Tree)**.** *A polynomial tree is a* 4*-tuple* $\langle V, E, r, l\rangle$*, consisting of:*

- *the finite set* V *of nodes,*
- *the set* $E \subseteq V \times V$ *of edges,*
- *the root node* $r \in V$*, and*
- *the labeling function* $l : V \to P \cup \mathbb{Z} \cup \{\, add, mul \,\}$*,*

where the graph (V, E) *is an ordered full binary tree rooted in* r*,* P *is a set of integer variables, and* l *respects the following rules:*

- *If* $v \in V$ *is a leaf,* $l(v) \in P \cup \mathbb{Z}$*.*
- *If* $v \in V$ *is an inner node,* $l(v) \in \{\, add, mul \,\}$*.*

Polynomial trees are representations of polynomials over integer variables, only using the operations addition (*add*) and multiplication (*mul*). Every node of a polynomial tree can be associated to a corresponding polynomial:

Definition 4.2.2 (Polynomial Tree Evaluation)**.** *Let* $T = \langle V, E, r, l\rangle$ *be a polynomial tree. The evaluation function* $p_T : V \to Pol$ *maps each node* $v \in V$ *to a polynomial, where* p_T *is defined recursively as follows:*

- *For a leaf* $v \in V$*,* $p_T(v) = l(v)$*.*
- *For an inner node* $v \in V$ *with the left child* v_l *and the right child* v_r*, we set:*

$$p_T(v) = \begin{cases} p_T(v_l) + p_T(v_r) & \text{if } l(v) = add \\ p_T(v_l) \cdot p_T(v_r) & \text{if } l(v) = mul \end{cases}$$

We use the notation $p(T)$ *as a shorthand for* $p_T(r)$*.*

With polynomial trees, we can now differentiate between different evaluation orders for the same polynomial. Figure 4.3 shows three different polynomial trees, which all evaluate to an equivalent polynomial.

By joining two polynomial trees with a relation operator, we obtain a representation of arithmetic constraints:

Definition 4.2.3 (Constraint Tree)**.** *A constraint tree is a* 3*-tuple* $\langle T_l, T_r, \sim\rangle$*, consisting of:*

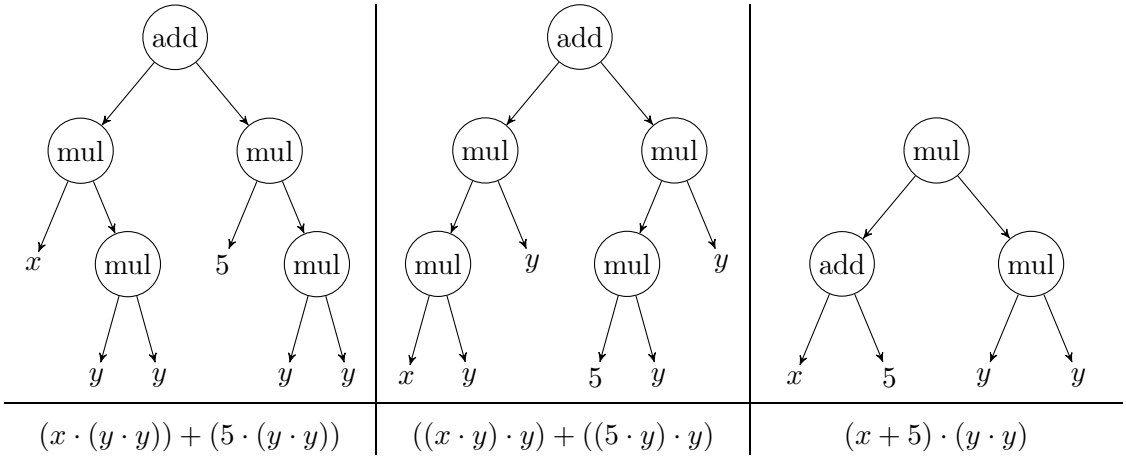


Figure 4.3: Different polynomial trees with $p(T) \equiv xy^2 + 5y^2$

- *a left polynomial tree* T_l
- *a right polynomial tree* T_r*, and*
- *a relation symbol* $\sim \in \{=, \neq, <, \leq, >, \geq\}$.

Formally, a constraint tree does not have a tree-like structure itself. Nevertheless, we call it a tree because we can view it as a binary tree that consists of a root node, labeled with $\sim$, and its two children, the subtrees T_l and T_r.

In analogy to the polynomial trees, we define an evaluation function p that maps a constraint tree to its arithmetic constraint:

Definition 4.2.4 (Constraint Tree Evaluation)**.** *The evaluation function p on polynomial trees can be expanded to constraint trees as follows: For a constraint tree $T_c = \langle T_l, T_r, \sim \rangle$, we set $p(T_c)$ to the inequality:*

$$p(T_c) := p(T_l) \sim p(T_r)$$

In Algorithm 4, the generation of constraint trees is the first step towards an encoding into bitvector logic. The REWRITE function receives an arithmetic constraint c and outputs a constraint tree T_c such that $p(T_c) \equiv c$. The set of all constraint trees is then denoted by C.

Although the only functional requirement for REWRITE is that REWRITE$(c) \equiv c$ for every constraint c, the choices made by REWRITE can have a significant effect on the performance of the bitvector solver. As an example, consider again the polynomial trees in Figure 4.3. The polynomial tree in the middle contains four multiplication nodes, which would be translated into four different `bvmul` applications. Three of the four multiplication nodes have two non-constant children, which would generate at least three full-blown multiplication networks.

The left polynomial tree also consists of three multiplication nodes, but the two lowermost multiplication nodes have the same children. More precisely, the subtrees rooted in these nodes are completely identical. Hence, it would be sufficient to create only a single multiplication network for the multiplication $y \cdot y$, and two multiplication networks for the multiplication nodes in the layer above. Among those, one node has a constant child (labeled with 5), which leads to a simplified network.

The third depicted polynomial tree is substantially different from the others. It makes use of a factorization of the polynomial $xy^2 + 5y^2$, such that only two multiplication nodes are required.

Presumably, this polynomial tree would lead to the most efficient SAT encoding among the given three examples. However, this locally optimal choice might not be globally optimal as well. As soon as other polynomials are encoded in addition (e.g. xy^2-2y), one of the other two polynomial trees might lead to better results as they might contain subtrees that are shared by newly added polynomial trees.

```
1: function REWRITE(constraint c = p ∼ 0)
2:     p′ ← p′ − CONSTANTPART(p)
3:     q′ ← −CONSTANTPART(p)
4:     ∼′←∼
5:     if LEADINGCOEFFICIENT(p′) < 0 then
6:         p′, q′ ← (p′ · (−1)), (q′ · (−1))
7:         ∼′← TURNAROUND(∼)
8:     end if
9:     return ⟨POLYTREE(p′), POLYTREE(q′), ∼′⟩
10: end function
11: function POLYTREE(polynomial p)
12:     SORTTERMS(p)
13:     if COUNTTERMS(p) > 1 then
14:         t ← LASTTERM(p)
15:         return NODE(add, POLYTREE(p − t), POLYTREE(t))
16:     end if                                          ▷ p only consists of one term
17:     c ← LEADINGCOEFFICIENT(p)
18:     if p = c then
19:         return NODE(c)
20:     end if                                          ▷ p is not constant
21:     if c ≠ 1 then
22:         return NODE(mul, POLYTREE(c), POLYTREE(t/c))
23:     end if                                  ▷ p is a monomial (no coefficient left)
24:     x^n ← FIRSTVARIABLEANDEXPONENT(p)
25:     if COUNTVARIABLES(p) > 1 then
26:         return NODE(mul, POLYTREE(x^n), POLYTREE(t/x^n))
27:     else if n > 1 then                          ▷ x is the only variable in p
28:         return NODE(mul, POLYTREE(x), POLYTREE(x^(n−1)))
29:     else                                                          ▷ p = x
30:         return NODE(x)
31:     end if
32: end function
```

Algorithm 5: The function rewrite

For our implementation, we decided for a rather simple constraint tree generation technique, which is illustrated in Algorithm 5. A given inequality $p \sim 0$ is transformed into an equivalent inequality $p' \sim' q'$ by moving the constant part of p to the right side and optionally multiplying by -1 to ensure that the leading coefficient of p' is positive. We then create polynomial trees for p' and q' by calling the recursive POLYTREE function.

The POLYTREE function sorts all terms of the provided polynomial according to a fixed ordering on monomials. Each term is rewritten on its own, starting from the right. After separating the coefficient, the monomial of the term is decomposed into a product of univariate monomials x^n, which becomes a chain of multiplication nodes. Figure 4.4 shows the generated constraint tree for the constraint $c = 3x^2y + x^2 + 4 < 0$.

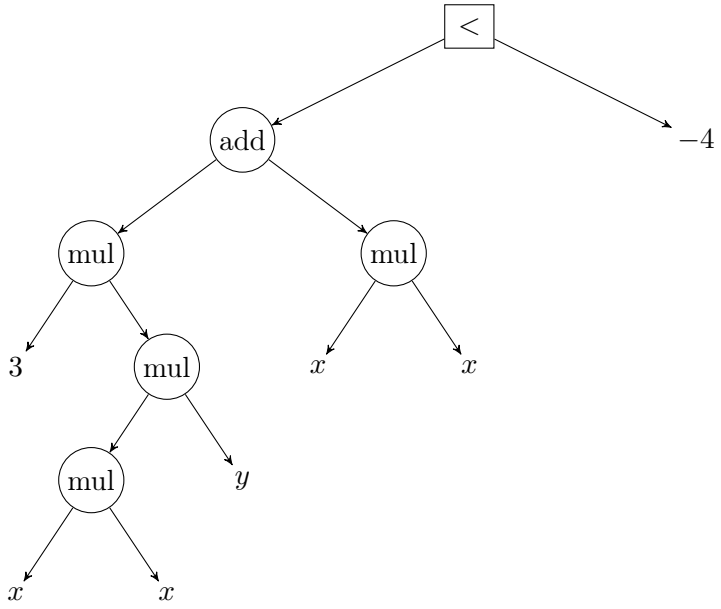


Figure 4.4: The generated constraint tree for $c : 3x^2y + x^2 + 4 < 0$

4.3 Encoding to Bitvector Arithmetic

The decision procedures BVModule.decide (Algorithm 3) and IntBlastModule.decide (Algorithm 4) share a lot of similarities. The `BVModule` translates bitvector variables into vectors of propositional variables (or, more generally, vectors of propositional formulas). Based on this, more complex bitvector terms and bitvector constraints can be reproduced on the SAT level. Analogously, the `IntBlastModule` reduces integer arithmetic to bitvector arithmetic by encoding integer variables as bitvector variables.

However, solving integer arithmetic by bitvector arithmetic bears two new important challenges: Firstly, integer variables range over an infinite domain, whereas bitvector variables have a fixed width and can thereby only attain finitely many different values. This does not only make the reduction incomplete, but also leaves the question how the widths should be chosen.

Secondly, the relation between integers and bitvectors is less natural than the relation between bitvectors and vectors of propositional variables. In Section 4.3.1, we formalize the concept of annotated bitvector terms, which establishes a connection between integers and bitvectors. Section 4.3.2 illustrates how the createFreshIntegerMapping function maps integers to annotated bitvector terms, and Section 4.3.3 demonstrates how the encode function applies this mapping to encode constraint trees into bitvector logic.

4.3.1 Annotated Bitvector Terms

It is a natural idea to encode bitvectors as vectors of propositional variables. Integer variables, on the other hand, have no such canonical representation as bitvectors. The bitvector literal `1111`, for example, might encode the number 15, but it could also refer to -1 or even a completely different number. This gives rise to the concept of annotated bitvector terms, which are a coupling of a bitvector expression and a specification of its semantics. We call such a specification a *bitvector annotation*.

The most fundamental property of a bitvector annotation is the signedness: We differentiate

between signed (s) and unsigned (u) expressions. Depending on the signedness, different ranges of integers can be represented, as depicted in Table 4.1.

Width	Signedness	Min. value	Max. value
4	u	0 (0000)	15 (1111)
4	s	-8 (1000)	7 (0111)
8	u	0 (00000000)	255 (11111111)
8	s	-128 (10000000)	127 (01111111)
n	u	0 (0 ... 0)	$2^n - 1$ (1 ... 1)
n	s	-2^{n-1} (10 ... 0)	$2^{n-1} - 1$ (01 ... 1)

Table 4.1: Representable ranges by width and signedness

So far, we have to choose a width and a signedness for each integer variable we want to encode into bitvector logic. Although we might as well use a fixed width and a fixed signedness for all variables, this approach allows us to react to different variable ranges: For many problems in integer arithmetic, the ranges of the variables in the formulas is bounded by some lower or upper value, e.g. by a global constraint $x > c$. If we encounter a constraint like $x \geq 0$, we know that we can encode x using unsigned semantics, such that we do not waste a bit for encoding the sign of x.

When looking at the ranges in Table 4.1, it becomes obvious that all possible ranges include the number 0. This might not be optimal: Consider, for example, an integer variable x that is bounded by $x \geq 260$ and $x < 264$. Then x ranges over an interval of only 4 elements. Though, we would need a width of $n = 9$ to cover this range using unsigned semantics, which seems to be very inefficient. To cope with these situations, we enrich bitvector annotations by a corrective constant called *offset*. When converting from bitvector values to integer values, the offset is added at the end of the conversion. Table 4.2 illustrates this effect.

Width	Signedness	Offset	Min. value	Max. value
4	u	0	0 (0000)	15 (1111)
4	s	0	-8 (1000)	7 (0111)
4	u	37	37 (0000)	52 (1111)
4	s	37	29 (1000)	44 (0111)
n	u	k	k (0 ... 0)	$k + 2^n - 1$ (1 ... 1)
n	s	k	$k - 2^{n-1}$ (10 ... 0)	$k + 2^{n-1} - 1$ (01 ... 1)

Table 4.2: Representable ranges by width, signedness and offset

It should be mentioned that the encoding process can only benefit from a non-zero offset if the formulas fulfill certain properties. For now, we skip this aspect, which is discussed in more detail in Section 4.3.2, and define the notion of bitvector annotations more formally.

Definition 4.3.1 (Bitvector annotation)**.** *A bitvector annotation is a tuple* $a = \langle a.sgn, a.offset \rangle$ *with:*

- *a signedness bit* $a.sgn \in \{ u, s \}$ *and*
- *an integer* $a.offset \in \mathbb{Z}$*.*

By combining a bitvector term with an annotation, we obtain an annotated bitvector term.

Definition 4.3.2 (Annotated bitvector term)**.** *An annotated bitvector term of width* $n \in \mathbb{N}$ *is a tuple* $t = \langle b_{[n]}, a \rangle$*, where* $b_{[n]}$ *is a bitvector term of width* n *and* a *is a bitvector annotation.*

We may also use the notations t.bvterm for $b_{[n]}$, t.width for n, t.annotation for a, t.sgn for a.sgn and t.offset for a.offset. In the case that $b_{[n]}$ consists only of a bitvector variable, we also call t an annotated bitvector variable.

Algorithm 4 uses annotated bitvector terms for an internal representation of each node in a polynomial tree. Just like BVModule.decide maps every bitvector expression to a vector of propositional variables, IntBlastModule.decide maps every node of a polynomial tree to an annotated bitvector term. Note that the annotations only serve as an internal fixing of the semantics of the corresponding bitvector term. They do not appear explicitly in the bitvector formulas C_{BV}, but only serve as a basis for encoding all further constraints. For an annotated bitvector term t with $t.sgn = s$, for example, the bitvector term $t.bitvec$ is always treated with signed bitvector operators, whereas a term with $t.sgn = u$ is usually processed by unsigned bitvector operators.

As already indicated in Table 4.2, different bitvector annotations lead to different representable ranges:

Definition 4.3.3 (Induced range of a bitvector annotation)**.** *For a bitvector annotation a and a width $n \in \mathbb{N}$, the induced range* $\text{range}_n(a)$ *is defined as:*

$$\text{range}_n(a) := \begin{cases} \{\, a.offset, \ldots, a.offset + 2^n - 1 \,\} & \text{if } a.sgn = u \\ \{\, a.offset - 2^{n-1}, \ldots, a.offset + 2^{n-1} - 1 \,\} & \text{if } a.sgn = s \end{cases} \tag{4.2}$$

We can now formalize the relationship that is illustrated in Table 4.2. Given a width n and a bitvector annotation a, we obtain a bijection between the set $BitVec_n$ of bitvectors and the set $\text{range}_n(a)$ of integers.

Definition 4.3.4 (Decoding and encoding using bitvector annotations)**.** *For a bitvector annotation a and $n \in \mathbb{N}$, we define the decoding function $a.decode : BitVec_n \to \text{range}_n(a)$ by:*

$$a.decode_n(b) := a.offset + \begin{cases} \text{bv2nat}_n(b) & \text{if } a.sgn = u \\ \text{bv2int}_n(b) & \text{if } a.sgn = s \end{cases} \tag{4.3}$$

The inverse function $a.encode_n : \text{range}_n(a) \to BitVec_n$ with $a.encode_n := (a.decode_n)^{-1}$ can be computed by:

$$a.encode_n(i) = \begin{cases} \text{nat2bv}_n(i - a.offset) & \text{if } a.sgn = u \\ \text{int2bv}_n(i - a.offset) & \text{if } a.sgn = s \end{cases} \tag{4.4}$$

Informally, *a.sgn* decides whether bv2nat or bv2int is used for the conversion, and *a.offset* is the value that the zero-only bitvector $\lambda i \in \{\, 0, \ldots, n-1 \,\}.0$ is mapped to.

In the following, we present how the `IntBlastModule` makes use of annotated bitvector terms in the functions createFreshIntegerMapping and encode.

4.3.2 Integer Mapping Creation

Let us again recall the encoding from bitvector arithmetic to propositional logic that is presented in Algorithm 3. The encoding process comprises two important steps. First, a bitvector mapping e is created, which maps every bitvector term to a vector of propositional variables, and every bitvector constraint to a single propositional variable. Essentially, e defines the logical

connection between bitvector interpretations and propositional interpretations. As soon as the bitvector mapping has been fixed, the original constraints are encoded into propositional formulas that ensure that the variables $V(e)$ indeed behave as it is dictated by the bitvector terms and constraints.

The reduction from integer arithmetic to bitvector arithmetic proceeds in a very similar way. First, a mapping between integer terms (i.e., polynomials over the integers) and annotated bitvector terms is created. In analogy to Chapter 3, we call such a mapping an *integer mapping*. As a second step, the created bitvector terms are linked by suitable bitvector formulas.

The following definition of an integer mapping is closely related to Definition 3.1.1.

Definition 4.3.5 (Integer mapping)**.** *An integer mapping is a mapping f, whose domain* $\mathrm{dom}(f)$ *consists of polynomials over the integers and integer constraints. For an integer constraint c, $f(c)$ is a propositional variable. For a polynomial p, $f(p)$ is an annotated bitvector variable.*

The function createFreshIntegerMapping in Algorithm 4 is responsible for the initial generation of an integer mapping. Despite its similarity to the createFreshBitvectorMapping function, there are differences that should be pointed out. First of all, the mapping returned by createFreshIntegerMapping is only defined on the integer variables appearing in the input formulas. For any other polynomial p, which is more complex than a simple integer variable, the returned integer mapping is undefined. It will be expanded ad-hoc by the `encode` function, as already indicated briefly in the optimizations in Section 3.3.

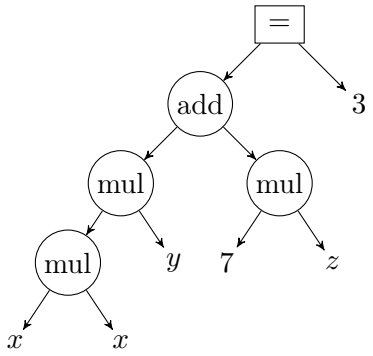
The most important novelty in the creation of an integer mapping is that a suitable width and annotation have to be chosen for each created bitvector. These decisions are based on properties which are extracted from the set of all constraint trees upfront. Algorithm 6 shows the implemented procedure.

```
 1: function IntBlastModule.createFreshIntegerMapping(constraint trees C)
 2:     V ← ∅
 3:     N ← ∅
 4:     for all c ∈ C do
 5:         V ← V ∪ variables(c)
 6:         N ← N ∪ nonlinearVariables(c)
 7:     end for
 8:     e ← ⊥
 9:     for all v ∈ V do
10:         R ← extractRange(C, v)
11:         nl ← v ∈ N
12:         e(v) ← createAnnotatedBitvector(R, nl)
13:     end for
14:     return e
15: end function
```

Algorithm 6: IntBlastModule.createFreshIntegerMapping

By iterating over all constraint trees C, the variables appearing in C are collected. The function call variables(c) emits the set of all variables which occur in the labeling of the leaves of c. The set returned by nonlinearVariables(c) is a subset of variables(C). It is composed of all variables in c which are involved in a nonlinear multiplication. More specifically, this comprises all variables that appear as a descendant of some multiplication node whose other child is not a constant. We illustrate this with an example.

Example 4.3.1. *Consider the following constraint tree* $c = \text{REWRITE}(x^2y + 7z - 3 = 0)$.



The two leftmost multiplication nodes are nonlinear, as none of them has a constant as child. Consequently, x and y are nonlinear variables.
The rightmost multiplication node, on the other hand, is linear, as its left child is the constant 7. Thus, z is not a nonlinear variable. We obtain:

$$\text{VARIABLES}(c) = \{ x, y, z \}$$
$$\text{NONLINEARVARIABLES}(c) = \{ x, y \}$$

We differentiate between variables that only occur linearly and those that eventually occur nonlinearly. The reason for the distinction is located in the "offset" part of bitvector annotations. Essentially, our encoding of nonlinear multiplications, which is presented in the next section, is only efficient if all of its operands have a zero offset. Any involved variable that is encoded with a non-zero offset would require an intermediate step for eliminating the offset prior to the multiplication. The introduced overhead would annihilate all benefits from choosing a non-zero offset. Hence, it makes more sense to forbid a non-zero offset for all nonlinear variables. We come back to the reasons for this requirement in the description of the encoding of multiplication nodes in the following Section 4.3.3.

After collecting the set of all variables V in C, Algorithm 6 iterates over each variable $v \in V$. The function call $\text{EXTRACTRANGE}(C, v)$ extracts bounds for v from the constraint trees C: The set C is scanned for constraint trees $\langle T_1, T_2, \sim \rangle$ such that T_1 is only a leaf labeled with v and T_2 is a constant leaf. These constraint trees correspond to constraints of the form $v \sim c$ for some relational symbol $\sim$ and $c \in \mathbb{Z}$. In other words, c is a lower or an upper bound for v (or both, if $\sim$ is the equality relation). By combining all such bounds, a (possibly unbounded) interval R for v is obtained.

The function CREATEANNOTATEDBITVECTOR makes the actual decision about how to encode a variable v. As an argument, it receives the extracted bounds R for v and the information whether v occurs nonlinearly. In addition, it uses a global constant MAX_WIDTH which limits the number of bits used for encoding each variable. The function then picks a width n and a bitvector annotation a according to the following heuristic:

1. If R is unbounded, set $n :=$ MAX_WIDTH, $a.\mathit{offset} := 0$ and $a.\mathit{sgn} := s$.
   (a) If R is semi-positive, set $a.\mathit{sgn} := u$.
   (b) If nl is *false*, ensure that $\text{range}_n(a) \subset R$:
      i. If $\text{upperBound}(\text{range}_n(a)) > \text{upperBound}(R)$,
      set $a.\mathit{offset} := -(\text{upperBound}(\text{range}_n(a)) - \text{upperBound}(R))$.
      ii. If $\text{lowerBound}(\text{range}_n(a)) < \text{lowerBound}(R)$,
      set $a.\mathit{offset} := \text{lowerBound}(R) - \text{lowerBound}(\text{range}_n(a))$.
2. If R is bounded:
   (a) If nl is *false*, choose $a.\mathit{sgn} := u$, $a.\mathit{offset} := \text{lowerBound}(R)$ and n minimal such that $\text{upperBound}(R) \in \text{range}_n(a)$ (but at most MAX_WIDTH).
   (b) If nl is *true*:
      i. If R is semi-positive, set $a.\mathit{sgn} := u$, otherwise $a.\mathit{sgn} := s$.

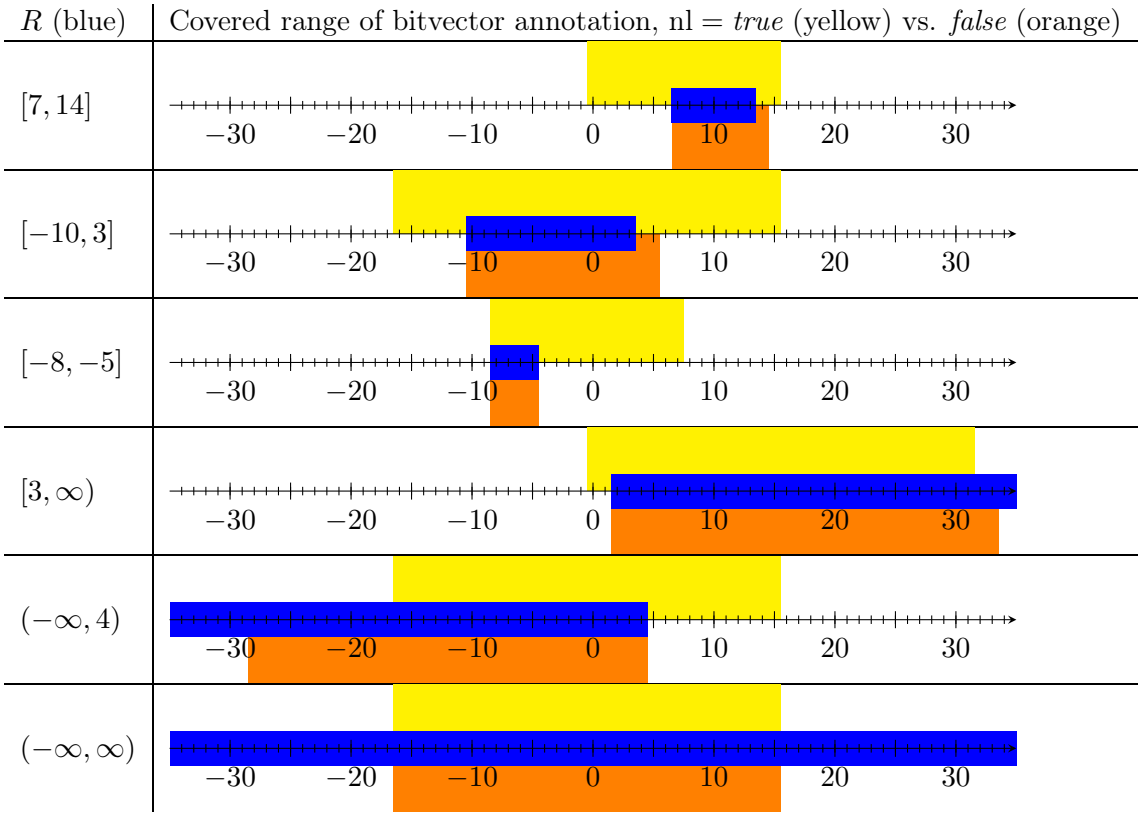


Table 4.3: Choices of CREATEANNOTATEDBITVECTOR for different values of R and nl, with the setting MAX_WIDTH = 5

ii. Choose n minimal such that $R \subseteq \text{range}_n(a)$ (but at most MAX_WIDTH).

To summarize, the procedure assigns width n and bitvector annotation a according to the following priorities: If nl is *true*, $a.\mathit{offset}$ is always 0. Moreover, the value for n must not exceed the constant MAX_WIDTH. If R contains negative numbers, a is chosen such that $\text{range}_n(a)$ also contains negative numbers. Under these conditions, n and a are picked to maximize $R \cap \text{range}_n(a)$. The width n is reduced as much as possible as long as $R \cap \text{range}_n(a)$ remains unchanged.

Table 4.3 visualizes $\text{range}_n(a)$ for the choices made by the described procedure for different inputs of R (left column, and painted in blue) and nl (yellow vs. orange bar). The width is limited by the constant MAX_WIDTH = 5, such that $\text{range}_n(a)$ can cover at most $2^5 = 32$ integers.

4.3.3 Translation of Constraint Trees

When Algorithm 4 reaches Line 4, all constraints of the input have been transformed into constraint trees. They determine a decomposition of each constraint into applications of addition, multiplication and a relational operator. For each variable v occurring in a leaf of any of the constraint trees, an encoding $e(v)$ as annotated bitvector variable has been fixed.

We now clarify the details of the ENCODE function, which builds on the generated constraint trees and the integer mapping e. For each constraint tree c, the ENCODE function creates a set of

constraints in bitvector logic, which express c on the basis of the integer mapping e. It proceeds as illustrated in Algorithm 7.

```
1: function IntBlastModule.encode(integer mapping e, constraint tree c)
2:     ⟨T_l, T_r, ∼⟩ ← c
3:     Φ ← ∅
4:     for all T ∈ { T_l, T_r } do
5:         for all n ∈ nodes(T) do                    ▷ in bottom-up order
6:             Φ ← Φ ∪ IntConstraints(e, n)
7:         end for
8:     end for
9:     Φ ← Φ ∪ IntConstraints(e, c)
10:    return Φ
11: end function
```

Algorithm 7: IntBlastModule.encode

The heart of encode is the function IntConstraints, which strongly resembles the function BVconstraints in Algorithm 3. It is applied to each node of c, starting with the leaves and proceeding in a bottom-up manner. Finally, IntConstraints is called for the whole constraint tree, i.e. the root node that is labeled with $\sim$.

The function IntConstraints takes a single node n of a constraint tree and encodes the expression $p(n)$, that is represented by n, into bitvector logic. For this purpose, a new annotated bitvector variable $e(p(n))$ is added to the integer mapping e. If n is an inner node, i.e. a node with two child nodes n_l and n_r, the IntConstraints function may assume that it has already been called on n_l and n_r, such that $e(p(n_l))$ and $e(p(n_r))$ already exist. It then outputs a set of bitvector formulas that restrict the values of $e(p(n))$ to the computation represented by n. In the case of an inner node, these formulas typically contain $e(p(n_l))$ and $e(p(n_r))$.

For convenience, we allow the notation $e(n)$ as an abbreviation for $e(p(n))$ for any polynomial tree node n. We now define the behavior of IntConstraints(e, n) for different nodes n. First, we assume that n is the node of a polynomial tree, such that $p(n)$ is a polynomial.

Encoding Variable Leaves

If n is a leaf labeled with an integer variable v, then an annotated bitvector term $e(v)$ already exists. It has previously been created by the function createFreshIntegerMapping (cf. Section 4.3.2).

In this case, we set:

$$\text{IntConstraints}(e, n) := \emptyset \tag{4.5}$$

Encoding Constant Leaves

If n is a leaf labeled with a constant $c \in \mathbb{Z}$, let a be the bitvector annotation defined by:

$$a.\mathit{offset} := 0 \tag{4.6}$$

$$a.\mathit{sgn} := \begin{cases} u & \text{if } c \geq 0 \\ s & \text{if } c < 0 \end{cases} \tag{4.7}$$

Then a fresh bitvector $b_{[m]}$ is created, where m is the smallest natural number such that $c \in \text{range}_m(a)$. The function adds the entry $e(n) := \langle b_{[m]}, a \rangle$ to e and returns the constraint:

$$\textsc{IntConstraints}(e, n) := \{\, b_{[m]} = a.\mathit{encode}_m(c) \,\} \tag{4.8}$$

Encoding Addition Nodes

Now, let n be a node that is labeled with *add* and has the left and right child nodes n_l and n_r. As stated already, we can assume that $t_l := e(n_l)$ and $t_r := e(n_r)$ are already defined. It is our goal to create an annotated bitvector term $e(n)$ that is constrained to model the sum of the annotated bitvector terms t_l and t_r. In principle, this can be achieved by a constraint like:

$$e(n).\mathit{bvterm} = \texttt{bvadd}(t_l.\mathit{bvterm}, t_r.\mathit{bvterm}) \tag{4.9}$$

However, there are several aspects that need to be handled. First of all, the annotations of t_l and t_r may use a different signedness. Although `bvadd` can be used both for signed and unsigned bitvectors, either both or none of its arguments should be signed. Secondly, the bitvector terms $t_l.\mathit{bvterm}$ and $t_r.\mathit{bvterm}$ can be of different widths, which is not permitted by the `bvadd` operator. On top of that, we need to ensure that the width of $e(n).\mathit{bvterm}$ is big enough to store the sum of $t_l.\mathit{bvterm}$ and $t_r.\mathit{bvterm}$. Otherwise, the `bvadd` operator might produce an overflow, which is of course not desired in the context of integer arithmetic.

We handle these issues by using three steps: First, we pick a bitvector annotation a for $e(n)$. For the signedness, we use an unsigned annotation if both t_l and t_r are unsigned, and a signed annotation otherwise. Second, a width m for $e(n)$ is chosen, such that $\text{range}_m(a)$ is always big enough to store the sum of the two summands. Third, both bitvector terms of t_l and t_r are extended to the width m and joined by the operator `bvadd`.

In the following, we explain these steps in more detail.

1. Let a be the bitvector annotation defined by:

$$a.\mathit{sgn} := \begin{cases} u & \text{if } t_l.\mathit{sgn} = u \text{ and } t_r.\mathit{sgn} = u \\ s & \text{if } t_l.\mathit{sgn} = s \text{ or } t_r.\mathit{sgn} = s \end{cases} \tag{4.10}$$

$$a.\mathit{offset} := t_l.\mathit{offset} + t_r.\mathit{offset} \tag{4.11}$$

   Note that we can simply set the offset of $e(n)$ to the sum of the offsets of t_l and t_r, such that we do not have to handle any offset on the formula level.

2. We create the width m by the following rules:

   - If $t_l.\mathit{sgn} = t_r.\mathit{sgn}$, we choose:

$$m := \max\{\, t_l.\mathit{width}, t_r.\mathit{width} \,\} + 1 \tag{4.12}$$

- Otherwise, $t_l.sgn \neq t_r.sgn$. Without loss of generality, let $t_l.sgn = s$ and $t_r.sgn = u$. Then we set:

$$m := \max\{\, t_l.width, t_r.width + 1 \,\} + 1 \tag{4.13}$$

  Here, 1 is added to $t_r.width$ to account for the conversion from an unsigned to a signed value. Any unsigned value can be converted to a signed value by prepending a constant 0 bit.

3. A fresh bitvector variable $b_{[m]}$ is created. The bitvector terms of t_l and t_r are expanded to width m by setting $t'_l := \text{expand}_m(t_l)$ and $t'_r := \text{expand}_m(t_r)$, where the expand function is defined by:

$$\text{expand}_m(t) := \begin{cases} \texttt{zero\_extend}_{m-t.width}(t.bvterm) & \text{if } t.sgn = u \\ \texttt{sign\_extend}_{m-t.width}(t.bvterm) & \text{if } t.sgn = s \end{cases} \tag{4.14}$$

   Finally, the entry $e(n) := \langle b_{[m]}, a \rangle$ is added to e, and the following constraint is returned:

$$\textsc{IntConstraints}(e, n) := \{\, b_{[m]} = \texttt{bvadd}(t'_l, t'_r) \,\} \tag{4.15}$$

Encoding Multiplication Nodes

The encoding of multiplication nodes is similar to the previously shown encoding of addition nodes. Let n be a node of a polynomial tree that is labeled with *mul* and has the left and right child nodes n_l and n_r. Again, we abbreviate $t_l := e(n_l)$ and $t_r := e(n_r)$.

The IntConstraints function performs the same three steps as for addition, namely the generation of a bitvector annotation, the computation of a sufficiently large width and the composition to a `bvmul` term using extension operators. However, the behavior differs in the computation of the width and, most importantly, in the handling of offsets.

We first regard the case that $t_l.offset = t_r.offset = 0$. Then we obtain the beforementioned three steps:

1. Let a be the bitvector annotation defined by:

$$a.sgn := \begin{cases} u & \text{if } t_l.sgn = u \text{ and } t_r.sgn = u \\ s & \text{if } t_l.sgn = s \text{ or } t_r.sgn = s \end{cases} \tag{4.16}$$

$$a.offset := 0 \tag{4.17}$$

2. Set the width m to:

$$m := t_l.width + t_r.width \tag{4.18}$$

   Without giving a proof here, this width is sufficient and optimal regardless of the signedness of t_l and t_r. It can be derived from the definition of range_m by estimations over the minimum and maximum value representable by t_l and t_r.

3. A fresh bitvector variable $b_{[m]}$ is created and $e(n) := \langle b_{[m]}, a \rangle$ is added to e. Again, we set $t'_l := \text{expand}_m(t_l)$ and $t'_r := \text{expand}_m(t_r)$ and return:

$$\textsc{IntConstraints}(e, n) := \{\, b_{[m]} = \texttt{bvmul}(t'_l, t'_r) \,\} \tag{4.19}$$

A handling of non-zero offsets in the encoding of multiplication nodes requires significantly more effort than for addition nodes: Assume that t_l and t_r have an unsigned annotation and width w. Let $\mathfrak{I}$ be an interpretation which assigns $t_l.bvterm$ a bitvector value $\mathfrak{I}(t_l.bvterm) := b_l \in BitVec_w$ and $t_r.bvterm$ a value b_r. Then t_l represents the integer value $\text{bv2nat}_w(b_l) + t_l.offset$, and t_r represents $\text{bv2nat}_w(b_r) + t_r.offset$.

We would now have to find a bitvector term $b_{[w]}$ and an offset $o \in \mathbb{Z}$ such that, independent of $\mathfrak{I}$, the equation holds:

$$\text{bv2nat}_w(\mathfrak{I}(b_{[w]})) + o = (\text{bv2nat}_w(b_l) + t_l.offset) \cdot (\text{bv2nat}_w(b_r) + t_r.offset) \tag{4.20}$$

An elegant solution can only be obtained for a special case: Assume that n_l is a leaf labeled with a constant c, and that t_r is an arbitrary annotated bitvector term (with a non-zero offset allowed). By our construction, we have $t_l.offset = 0$. Then we can encode the node n using the bitvector annotation a with:

$$a.offset := c \cdot t_r.offset \tag{4.21}$$

The signedness of a is chosen as above, and steps 2 and 3 are identical to the previously regarded case.

For the general case of arbitrary annotated bitvector terms t_l and t_r with non-zero offset, there is no such direct construction of $e(n)$. Roughly speaking, we would have to encode $t_l.offset$ and $t_r.offset$ into bitvector literals, and proceed with the terms $\texttt{bvadd}(t_l.bvterm, \text{int2bv}(t_l.offset))$ and $\texttt{bvadd}(t_r.bvterm, \text{int2bv}(t_r.offset))$. Thereby, the offset is eliminated by encoding it into a bitvector addition.

However, due to the overhead of such a construction, which destroys all offset-related benefits, we instead ensure by our construction that such a constellation does not occur: Wherever a multiplication node n with two non-constant children exists, all variables appearing in the leaves below n are treated as nonlinear, such that CREATEFRESHINTEGERMAPPING encodes them with an offset of 0. This property is retained among all descendants of n, which makes it possible to encode n without the need to handle any offsets.

Encoding Relational Operators

At the end of the ENCODE function, the root of the constraint tree $c = \langle T_l, T_r, \sim \rangle$ is encoded. As $p(c)$ is not a polynomial, but an integer constraint, we set $e(p(c))$ to a fresh propositional variable R instead of an annotated bitvector term.

By our construction of constraint trees, we may assume that T_r only consists of a single, constant node. Let k be this constant. For the left subtree T_l, let r be the root of T_l and $t_l := e(r)$. Let further $a := t_l.annotation$ and $m := t_l.width$ be the annotation and width of t_l.

The relational symbol $\sim$ is mapped to a bitvector operator bvop by the following table:

$\sim$	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
bvop ($t_l.sgn = u$)	$=$	`distinct`	`bvult`	`bvule`	`bvugt`	`bvuge`
bvop ($t_l.sgn = s$)	$=$	`distinct`	`bvslt`	`bvsle`	`bvsgt`	`bvsge`

We now encode the root of the constraint tree by:

$$\text{INTCONSTRAINTS}(e, c) := \{\, R \leftrightarrow \text{bvop}(t_l.bvterm, a.encode_m(k)) \,\} \tag{4.22}$$

In the case that $k \notin \text{range}_m(a)$, we know that k is above (or below) every possible integer value that is represented by t_l. Under this condition, we can evaluate $\sim$ directly and set IntConstraints(e, c) to either $\emptyset$ or $\{ \mathit{false} \}$.

4.4 Bounds from Interval Constraint Propagation

As shown in the previous section, the encode function encodes each polynomial tree node into an annotated bitvector term. In this process, it has to choose a width for every new annotated bitvector variable it creates. Naturally, the chosen widths increase from bottom to top nodes: For an addition node, the assigned width is at least the maximum of the widths of its children plus one. For a multiplication node, the width is set to the sum of the widths of its children.

Of course, this growth has a remarkable impact on the performance of the bitvector solver. Both addition and multiplication are rather costly operations (in particular in comparison to bit-level operations) and the number of generated CNF clauses grows nonlinearly with the operator width. Hence, it is desirable to reduce the assigned widths as much as possible.

The IntConstraints function calculates its widths very conservatively. In each node of the polynomial tree, it assumes that the bitvector expressions of its children may take an arbitrary value. Assume, for example, the constraints:

$$C = \{ x \geq 0, y \geq 0, x \leq 10, y \leq 3, x + y > 8 \} \tag{4.23}$$

Then the encoding process might encode x by an unsigned bitvector of width 4 and y by an unsigned bitvector of width 2. Consequently, a width of $\max\{4, 2\} + 1 = 5$ would be chosen for encoding the sum $x + y$. However, in a satisfying assignment for C, $x + y$ can be at most 13, such that a width of 4 would actually be sufficient.

A similar effect can also be observed with a much higher significance for the constraint set:

$$C = \{ x \geq 0, x \leq 250, y \geq 0, x^2 + y < 10, x^3 > 20 \} \tag{4.24}$$

The chooseFreshIntegerMapping function only uses very simple constraints to determine the widths that it generates. Although the constraint combination $\{ y \geq 0, x^2 + y < 10 \}$ effectively limits the value of x by 3, chooseFreshIntegerMapping is unable to detect such an effect and selects a width of 8 for encoding y. This results into 24 bits for encoding the polynomial x^3, while in fact 5 bits would suffice (since $2^5 = 32 > 3^3$).

To summarize, considerable improvements can be obtained by generating better bounds for the encoded polynomials. For this purpose, we enriched the `IntBlastModule` by a method called *Interval Constraint Propagation*, which is presented in the following sections.

4.4.1 Introduction to ICP

Interval Constraint Propagation [Dav87] is an algorithm that operates on a set of arithmetic equations C over a set of variables V. A function I is provided, which maps each $v \in V$ to a corresponding interval $I(v)$. The algorithm now successively applies propagation rules to contract each interval $I(v)$ to an interval $I'(v) \subseteq I(v)$, such that the constraints C have the same solutions in the intervals I' as in the intervals I.

For an intuition of how Interval Constraint Propagation works, we give a simple example. Consider the following input:

$$C = \{ x + y = 10, z = x - 2y \} \tag{4.25}$$
$$V = \{ x, y, z \} \tag{4.26}$$
$$I(x) = I(y) = [0, 255] \tag{4.27}$$
$$I(z) = (-\infty, \infty) \tag{4.28}$$

ICP originates from the domain of real numbers, but it can easily be adapted to integer numbers as well. For our purposes, we assume that x, y and z are variables ranging over $\mathbb{Z}$. Initially, the contracted intervals I' are set to $I' := I$. The procedure now repeatedly selects a *contraction candidate*, i.e. a constraint $c \in C$ and a contained variable $v \in V$, and tries to contract the interval $I'(v)$ using the constraint c. The choices are made by a heuristic and may be nondeterministic. On our example, ICP might apply the following steps:

1. Let $c : x+y = 10$ and $v = x$. The constraint c is solved for x, resulting in $x = 10-y$. From $y \in I'(y) = [0, 255]$ we can deduce $10 - y \in [-245, 10]$. Hence, the algorithm performs the contraction:
$$I'(x) := I'(x) \cap [-245, 10] = [0, 255] \cap [-245, 10] = [0, 10] \tag{4.29}$$

2. Let $c : z = x - 2y$ and $v = x$. Solving c for x, we get $x = z + 2y$. As we have $I'(z) = (-\infty, \infty)$, no further contraction can be obtained from this combination of c and v.

3. Let $c : x + y = 10$ and $v = y$. From $y = 10 - x$ and $I'(x) = [0, 10]$ we get $10 - x \in [0, 10]$ and contract $I'(y)$ to:
$$I'(y) := I'(y) \cap [0, 10] = [0, 255] \cap [0, 10] = [0, 10] \tag{4.30}$$

4. Let $c : z = x - 2y$ and $v = z$. The constraint c is already solved for z. From $x \in I'(x) = [0, 10]$ and $y \in I'(y) = [0, 10]$ we obtain $x - 2y \in [-20, 10]$ and set:
$$I'(z) := I'(z) \cap [-20, 10] = (-\infty, \infty) \cap [-20, 10] = [-20, 10] \tag{4.31}$$

At this point, no further contractions are possible and the algorithm terminates with the result:

$$I'(x) = [0, 10] \tag{4.32}$$
$$I'(y) = [0, 10] \tag{4.33}$$
$$I'(z) = [-20, 10] \tag{4.34}$$

For brevity, we leave out many technical details here, including the formal foundation of interval arithmetic and contraction rules, required adaptations and preprocessing to ensure termination, the generalization from equations to inequations, more efficient procedures for contraction, benefits from coupling ICP with SAT or linear arithmetic solvers, and heuristics for the choice of contraction candidates. For an in-depth discussion of ICP, in particular concerning the implementation in SMT-RAT, we refer to [Sch13].

In the context of our thesis, we can treat the `ICPModule` in SMT-RAT as a black box, which receives a set C of arithmetic constraints (including inequalities) and generates range constraints

over the variables in C. Alternatively, it may return *unsat* if the ICP algorithm contracts an interval to the empty set, or *sat* if the satisfiability of C is detected (e.g. by test points obtained from the ends of the intervals).

In general, the running time of ICP is hard to predict, as it depends on many implementation factors. However, we found it to perform very well and to contribute only negligibly to the overall running time of our algorithm.

4.4.2 Bound Generation for Polynomial Tree Nodes

The `IntBlastModule` uses Interval Constraint Propagation to generate bounds for each node of the polynomial trees that are encoded. As a side benefit, it may detect the satisfiability or unsatisfiability of the constraints inside the restricted bounds. In this case, the translation to bitvector logic can be skipped completely. Algorithm 8 depicts the improved decision procedure of the `IntBlastModule`.

```
 1: function IntBlastModule.decide(input formulas C_rcv)
 2:     C ← { rewrite(c) | c ∈ C_rcv }
 3:     e ← createFreshIntegerMapping(C)
 4:     C_ICP ← { decompose(c) | c ∈ C } ∪ { inBounds(e) }
 5:     icp ← ICP.decide(C_ICP)
 6:     if icp = sat then
 7:         return sat
 8:     else if icp = unknown then                          ▷ Run bitvector solver
 9:         C_BV ← ∅
10:         I' ← ICP.deducedBounds(C_ICP)
11:         for all c ∈ C do
12:             C_BV ← C_BV ∪ encode(e, I', c)
13:         end for
14:         if BVSolver.decide(C_BV) = sat then
15:             return sat
16:         end if
17:     end if                                              ▷ C_rcv ⊨ ¬ inBounds(e)
18:     C_pas ← C_rcv ∪ { ¬ inBounds(e) }
19:     return backend.decide(C_pas)
20: end function
```

Algorithm 8: IntBlastModule.decide (with ICP)

After the initialization of the integer mapping, constraints for ICP are generated by applying decompose to each generated constraint tree. The decompose function decomposes a constraint tree $c = \langle T_l, T_r, \sim \rangle$ into an equisatisfiable set of constraints, where each constraint contains at most one addition or multiplication operator. For this purpose, a fresh integer variable $s_{p(n)}$ is created for each node n of T_l and T_r (cf. Definition 4.2.4). Two nodes n_1, n_2 with $p(n_1) = p(n_2)$ share the same integer variable $s_{p(n_1)}$. The function decompose then returns the set:

$$\begin{aligned} \textsc{decompose}(\langle T_l, T_r, \sim \rangle) := &\{ \text{nodeConstraint}(n) \mid n \in \text{Nodes}(T_l) \cup \text{Nodes}(T_r) \} \\ &\cup \{ s_{p(T_l)} \sim s_{p(T_r)} \} \end{aligned} \tag{4.35}$$

For a polynomial tree node n with the child nodes n_l and n_r (if present) and the label $l(n)$, the function nodeConstraints returns the integer constraint:

$$\text{nodeConstraints}(n) := \begin{cases} s_{p(n)} = l(n) & \text{if } n \text{ is a leaf} \\ s_{p(n)} = s_{p(n_l)} + s_{p(n_r)} & \text{if } l(n) = add \\ s_n = s_{p(n_l)} \cdot s_{p(n_r)} & \text{if } l(n) = mul \end{cases} \tag{4.36}$$

Example 4.4.1. *Let T be the constraint tree $T :=$ REWRITE$(x + y^2 > 5)$. Then T is decomposed into the set:*

$$\begin{aligned} \text{DECOMPOSE}(T) = \{ & s_x = x, && s_y = y, && s_{y^2} = s_y \cdot s_y, \\ & s_{x+y^2} = s_x + s_{y^2}, && s_5 = 5, && s_{x+y^2} > s_5 \quad \} \end{aligned}$$

The decomposition constraints are collected for all constraint trees and passed to ICP. In addition, ICP receives the formula inBounds(e), which expresses the bounds that are introduced by the choice of the integer mapping. For a variable x with $e(x) = \langle b_{[n]}, a \rangle$ and $\text{range}_n(a) = [0, 255]$, for example, the constraints $x \geq 0$ and $x \leq 255$ are added to the ICP input C_{ICP}.

By construction, the set C_{ICP} is equisatisfiable to $C_{rcv} \cup \{ \text{inBounds}(e) \}$, which in turn is equisatisfiable to the later generated set C_{BV} of bitvector constraints. Hence, Algorithm 8 returns *sat* if C_{ICP} is satisfiable, and skips the call to the bitvector solver if C_{ICP} is unsatisfiable. If neither satisfiability nor unsatisfiability is detected, ICP is queried for its contracted intervals by the call ICP.DEDUCEDBOUNDS(C_{ICP}). The function returns I', as described in Section 4.4.1, i.e. a mapping from the variables in C_{ICP} to their contracted intervals.

As explained in the beginning of Section 4.4, the bounds from I' are used to reduce the widths that are needed to encode polynomial tree nodes. As such, I' is passed to the call to ENCODE in Line 12 of Algorithm 8. In the following, we describe how the ENCODE method is modified to utilize I' in its choices.

4.4.3 Applying Inferred Bounds for Width Reduction

As we have seen, the ENCODE function of the `IntBlastModule` calls the function INTCONSTRAINTS on each node n. For the width reduction, we are mainly concerned with the inner nodes of the polynomial trees, i.e. the nodes that represent addition or multiplication operations. On such a node n, the INTCONSTRAINTS function operates in three steps. First, an annotation a is chosen. Second, the new width w is calculated conservatively from the width of the child node encodings. Third, the new bitvector term is composed by extending the bitvector terms of the children to width w and joining them with `bvadd` or `bvmul`. The full result is then stored as $e(n)$.

Now, using the information from ICP, we know that in any satisfying assignment for $C_{rcv} \cup \{ \text{inBounds}(e) \}$, the value of the polynomial $p(n)$ is inside the interval $I'(S_{p(n)})$. Hence, we are able to compute the minimum width $\hat{w}$ that is sufficient to represent any value in $I'(S_{p(n)})$ under the annotation a:

$$\hat{w} := \min \left\{ m \in \mathbb{N} \;\middle|\; I'(S_{p(n)}) \subseteq \text{range}_m(a) \right\} \tag{4.37}$$

At first sight, it may be tempting to replace w by $\hat{w}$ in step 2 of the INTCONSTRAINTS call.

1: **function** IntBlastModule.encode(integer mapping e, intervals I', constraint tree c)
2: $\langle T_l, T_r, \sim \rangle \leftarrow c$
3: $\Phi \leftarrow \emptyset$
4: **for all** $T \in \{ T_l, T_r \}$ **do**
5: **for all** $n \in \text{NODES}(c)$ **do** ▷ in bottom-up order
6: $\Phi \leftarrow \Phi \cup \text{IntConstraints}(e, n)$
7: $\Phi \leftarrow \Phi \cup \text{shrink}(e, I', n)$
8: **end for**
9: **end for**
10: $\Phi \leftarrow \Phi \cup \text{IntConstraints}(e, c)$
11: **return** Φ
12: **end function**

Algorithm 9: IntBlastModule.encode (with ICP)

However, due to the overflow behavior of `bvadd` and `bvmul`, this can lead to incorrect results, as the following example demonstrates.

Example 4.4.2. *Let* $C_{rcv} := \{ 0 \leq x, x \leq 7, 0 \leq y, y \leq 7, x \cdot y < 10 \}$. *Assume that* CreateFreshIntegerMapping *creates bitvector variables* $b_{[3]}$ *for* x *and* $c_{[3]}$ *for* y, *both with an unsigned and offset-free annotation.*

Without ICP, the polynomial tree node n *with* $p(n) = x \cdot y$ *would be encoded using* $3 + 3 = 6$ *bits. This is inefficient, as* $x \cdot y$ *is constrained to be less than* 10. *Therefore, ICP might compute an interval* $I'(S_{x \cdot y}) = [0, 9]$. *From this, we can calculate the optimized width* $\hat{w} = 4$ *for the encoding of* $x \cdot y$.

However, if the function IntConstraints *used* $\hat{w}$ *instead of* w *for the width of* $e(n)$, *we would obtain a constraint for* $e(n)$ *like:*

$$m_{[4]} = \texttt{bvmul}(\texttt{zero\_extend}_1(b_{[3]}), \texttt{zero\_extend}_1(c_{[3]})) \tag{4.38}$$

The above equation no longer models the correct semantics of arithmetic multiplication, due to the modular semantics of `bvmul`. *Assume, for example, an assignment* $\mathfrak{I}$ *with* $\mathfrak{I}(b) = \texttt{101}$ *(*$\hat{=}5$*),* $\mathfrak{I}(c) = \texttt{111}$ *(*$\hat{=}7$*) and* $\mathfrak{I}(m) = \texttt{0011}$ *(*$\hat{=}3$*). Then* $\mathfrak{I}$ *is a model of Equation 4.38, although clearly* $5 \cdot 7 \neq 3$.

The problem in Example 4.4.2 is that we need the conservative width w to ensure that the `bvmul` operation creates no overflow. Multiplying `101` and `111` after extending to $w = 6$ leads to the result `100011`. By choosing the width $\hat{w}$ instead, we lose track of the most-significant `1` appearing in the correctly computed result.

To avoid this wrong behavior, we use a two-step approach inside the revised ENCODE function, which is depicted in Algorithm 9. Again, we call IntConstraints on every node n, which creates $e(n)$ with a conservatively chosen width. Afterwards, the width is reduced by the call to the function SHRINK in Line 7. This method is allowed to overwrite $e(n)$ by a new annotated bitvector term and add further constraints to the set Φ.

Algorithm 10 shows how the SHRINK function proceeds on a polynomial tree node n. It extracts the ICP interval for $S_{p(n)}$ and compares it to the conservatively chosen range. If the ICP interval is smaller, the optimized width $\hat{w}$ is computed. If $\hat{w}$ is smaller than w, the actual shrinking is performed. For this purpose, a fresh bitvector $x_{[\hat{w}]}$ of width $\hat{w}$ is created. It receives

the same annotation as the previous $e(n)$ and thus only differs in its width. The new variable $x_{[\hat{w}]}$ is then constrained to the $\hat{w}$ least significant bits of $t_{[w]}$, which is the old bitvector term for $e(n)$.

Up to this point, the issue of Example 4.4.2 has not been addressed yet. This problem is mitigated in Lines 12 to 16. The formulas that are added to Φ require that the numerical representation of $t_{[w]}$ and $x_{\hat{w}}$ are equal. For an unsigned annotation, this corresponds to the constraint that only bits of value `0` are removed; for a signed annotation, all removed bits have to equal the sign bit (i.e., the most significant bit) of $x_{\hat{w}}$.

```
 1: function IntBlastModule.shrink(int. mapping e, intervals I', polyn. tree node n)
 2:     old ← e(n)
 3:     t_[w] ← old.bvterm
 4:     a ← old.annotation
 5:     I ← I'(S_p(n))
 6:     if I ⊂ range_w(a) then
 7:         ŵ ← min { m ∈ ℕ | I ⊆ range_m(a) }
 8:         if ŵ < w then                                  ▷ shrinking is possible
 9:             x_[ŵ] ← createFreshBitvectorVariable(ŵ)
10:             e(n) ← ⟨x_[ŵ], a⟩                          ▷ overwrite value e(n)
11:             Φ ← { x_[ŵ] = extract_{ŵ−1,0}(t_[w]) }
12:             if a.sgn = u then
13:                 Φ ← Φ ∪ { t_[w] = zero_extend_{ŵ−w}(x_[ŵ]) }
14:             else if a.sgn = s then
15:                 Φ ← Φ ∪ { t_[w] = sign_extend_{ŵ−w}(x_[ŵ]) }
16:             end if
17:             return Φ
18:         end if
19:     end if
20:     return ∅
21: end function
```

Algorithm 10: IntBlastModule.shrink

Example 4.4.3. *For the multiplication $x \cdot y$ in Example 4.4.2, the revised encoding process creates a bitvector term $x_{[4]}$ with the constraints:*

$$m_{[6]} = bvmul(zero\_extend_3(b_{[3]}), zero\_extend_3(c_{[3]})) \tag{4.39}$$

$$x_{[4]} = extract_{3,0}(m_{[6]}) \tag{4.40}$$

$$m_{[6]} = zero\_extend_2(x_{[4]}) \tag{4.41}$$

With these mechanism, the remaining parts of the encoding algorithm can make use of the more compact representation $x_{[\hat{w}]}$ instead of $t_{[w]}$, without losing the correctness of the construction.

4.5 SMT-Compliance

In Section 2.2 we introduced the notion of SMT-compliance. A module is called SMT-compliant if it supports incrementality, infeasible subset generation and backtracking. Both of the modules

that are subject of this thesis, namely the `BVModule` and the `IntBlastModule`, are implemented in an SMT-compliant fashion. While Algorithm 3 can be made SMT-compliant without major problems, there are important aspects to consider for the SMT-compliance of Algorithm 4. We use this section to address the arising challenges and our implemented solutions.

Incrementality And Backtracking

The requirement of incrementality states that new formulas should be addable to C_{rcv} such that their addition does not demand a full recomputation during the next satisfiability check. In principle, this can easily be achieved: Whenever a formula c is added to C_{rcv}, its decomposition is added to C_{ICP}, its constraint tree is added to C_{BV} after being processed by the ENCODE function, and c is added to the set C_{pas}, that is passed to the backend.

However, problems can occur regarding the integer mapping: The integer mapping is created by the function CREATEFRESHINTEGERMAPPING on the basis of all constraints received so far. For each new integer variable v, the global analysis process collects simple bounds for v from all received constraints, and detects whether v occurs only linearly or also nonlinearly. Based on these two properties, a decision about the encoding $e(v)$ is made.

A conflict may arise if a new formula c contains an already encoded variable v. Assume, for example, that during previous decidability checks the variable v has been encoded by $e(v) = \langle b_{[8]}, a\rangle$ with some bitvector term $b_{[8]}$ and an annotation a with $a.sgn = u$ and $a.offset = 200$. If v is contained nonlinearly in c, the annotation a is no longer feasible, as it uses a non-zero offset. In such an event, we need to regenerate $e(v)$, considering all received formulas (including c), and reencode all previously encoded constraints containing v.

A weaker variant of this event would be the addition of the constraint $c : v < 204$. If this constraint had previously been known, $e(v)$ could have been encoded using a bitvector variable of width 2 instead of 8, which would have resulted in a better performance of the bitvector solver. Regarding backtracking, the inverse effect is also possible: For the set $C_{rcv} = \{\, v \geq 0, v \leq 3 \,\}$, for example, the variable v is usually encoded by a bitvector of width 2. If the constraint $v \leq 3$ is removed, this choice becomes suboptimal, as it only covers a small part of the search space.

The effects described in the previous paragraph can be handled by different strategies. One possibility is to ignore these events and to leave $e(v)$ unchanged. This requires no reencoding, but it lowers the chances that the bitvector solver finds a solution. Alternatively, one could always recreate $e(v)$ and all affected formulas as soon as a better choice can be made. This makes it more likely that the bitvector search space contains a solution, but it is probably less performant due to a frequent reencoding. Our implementation is closer to the first approach: We only reencode if the covered range of $e(v)$ and the interval of possible values for v are disjoint (which would make the bitvector encoding infeasible).

The addition of ICP to the `IntBlastModule`, which is presented in Section 4.4, also bears the potential to interfere with incrementality: In Example 4.4.2, we examined the set $C_{rcv} = \{\, 0 \leq x, x \leq 7, 0 \leq y, y \leq 7, x \cdot y < 10 \,\}$. The last constraint made it possible for ICP to deduce that $x \cdot y \in [0, 9]$, such that we were able to shrink the bitvector term for $e(x \cdot y)$ to a smaller width. If the last constraint is now removed, the shrinking is no longer admissible. To handle this, we keep track of all shrunk encodings, and reencode all affected polynomials and constraints as soon as the updated ICP interval is no longer covered.

Infeasible Subset Generation

The generation of infeasible subsets is relevant whenever the `IntBlastModule` reports *unsat*. Two different scenarios are possible for this outcome:

1. The ICP module returns *unsat*, the backend returns *unsat*.
2. The ICP module returns *unknown*, the bitvector solver and the backend both return *unsat*.

All involved modules are capable of generating infeasible subsets. In the first scenario, let IS_{ICP} and $\mathrm{IS}_{backend}$ be the two corresponding infeasible subsets. The set IS_{ICP} may contain constraints from the formula $\mathrm{inBounds}(e)$, while $\mathrm{IS}_{backend}$ may contain the formula $\neg\,\mathrm{inBounds}(e)$. These bound-related formulas are removed from the sets, resulting in two sets IS'_{ICP} and $\mathrm{IS}'_{backend}$. Then the union $\mathrm{IS}_{IntBlast} := \mathrm{IS}'_{ICP} \cup \mathrm{IS}'_{backend}$ is an infeasible subset of C_{rcv}:

$$\begin{aligned}
\mathrm{IS}_{IntBlast} &\models \mathrm{IS}_{IntBlast} \cup \{\, \mathrm{inBounds}(e) \vee \neg\,\mathrm{inBounds}(e) \,\} \\
&\models \mathrm{IS}'_{ICP} \cup \mathrm{IS}'_{backend} \cup \{\, \mathrm{inBounds}(e) \vee \neg\,\mathrm{inBounds}(e) \,\} \\
&\models (\bigwedge \mathrm{IS}'_{ICP} \wedge \mathrm{inBounds}(e)) \vee (\bigwedge \mathrm{IS}'_{backend} \wedge \neg\,\mathrm{inBounds}(e)) \\
&\models (\bigwedge \mathrm{IS}_{ICP}) \vee (\bigwedge \mathrm{IS}_{backend}) \\
&\models \mathit{false}
\end{aligned}$$

The second case, in which the ICP module returns *unknown*, is more difficult: At first glance, one might use an infeasible subset IS_{BV} from the bitvector solver, replace each bitvector constraint by its original integer constraint, and return a union with $\mathrm{IS}_{backend}$ as above. Unfortunately, this is not sufficient. The main problem is that deductions from ICP are used for the construction of the bitvector constraints. Thus, each constraint c in the set IS_{BV} would have to be mapped not only to its original integer constraint, but also to the set of all constraints in C_{rcv} that implied an ICP deduction which in turn has been used in the generation of c.

Sadly, the ICP module in SMT-RAT currently does not link its deductions with the responsible original constraints. For this reason, we have to return a trivial infeasible subset in the second case, i.e. the full set of input constraints C_{rcv}.

4.6 Optimizations

We conclude this chapter by presenting some optimization aspects of our implementation.

Mapping to Terms Instead of Variables

We start with an idea that we already implemented with much success in the `BVModule`. So far, the definition of an integer mapping states that each integer variable is mapped to an annotated bitvector variable. By allowing arbitrary annotated bitvector terms instead of only variables, the number of created bitvector variables and bitvector equations can be reduced. This change creates the effect that the produced bitvector constraints may become very long and contain many shared subexpressions, but this can efficiently be handled by the structural hashing inside our bitvector module.

Lazy Encoding of Constants

The following optimization is devoted to the encoding of polynomial tree leaves that are labeled with constants. In Section 4.3.3 we proposed to encode a constant c into an annotated bitvector variable with zero offset and minimum width to store the value c. In most cases, the parent of the constant node is an addition or multiplication, such that the newly encoded constant has to be extended to another width. Instead, we could as well delay the encoding of a constant until a concrete bitvector representation is desired.

This concept does not only prevent an unnecessary step in the encoding of constants, but it also allows for a higher flexibility in the encoding of addition and multiplication. For example, a constant c can be added to an annotated bitvector term $t = \langle b, a \rangle$ by simply increasing $a.\mathit{offset}$ by c. A rule for multiplication with constants has already been presented in Section 4.3.3. In addition, ICP may deduce a point interval $[c, c]$ for some polynomial p, in which case we are also able to treat p as a constant and delay its encoding into bitvector logic.

Technically, we generalize the definition of an integer mapping by including $\mathbb{Z}$ in its codomain. Thereby, each polynomial is either represented as an annotated bitvector term or as an integer constant.

Eager encoding to bitvector logic

In the form presented in this chapter, the `IntBlastModule` acts as a theory module. It receives a set of integer constraints which do not contain any Boolean structure. This simplifies the implementation and allows a full exploitation of the ICP mechanism, which only works on conjunctions of integer constraints.

However, for a maximum flexibility in configuration, we implemented the `IntBlastModule` in a way that it can also handle formulas containing arbitrary Boolean connectives or constraints from other theories. The implementation idea is very similar to the proceeding in Algorithm 3, where every formula is replaced with its propositional skeleton.

As already discussed in Section 3.3, the bitvector module currently operates best in an eager mode. By switching to an eager encoding in the `IntBlastModule`, the internal bitvector solver is also operated in an eager fashion, such that we hope to achieve better performance results with this updated strategy composition.

Chapter 5

Evaluation

For an evaluation of the newly developed SMT-RAT modules, we compare the performance on benchmarks that originate from the SMT-LIB benchmark collections for QF_BV and QF_NIA. More specifically, the subset of benchmarks for the SMT-COMP 2015 [CDW15] has been chosen. We use the following sections to present the most important results and briefly comment on them.

5.1 SMT-COMP 2015 Results

SMT-RAT participated in the SMT-COMP 2015 with a prefinal version of the bitvector module. The results are listed in Table 5.1. All of the calculations in this chapter were performed on an Intel(R) Xeon(R) CPU E5-2609 with 2.40 GHz. For the results in Table 5.1, an available memory of 61440 MB was used.

The 14 errors in the computations of SMT-RAT could be tracked down to issues of an underlying module. Apart from that, it is apparent that the performance achieved by the competition version of SMT-RAT still leaves room for improvement. More recent results on the basis of the current version of the project can be found in the following section.

Solver	Errors	Corrects	CPU time	Not solved
Boolector	0	26260	647474.64	154
CVC4	0	26001	1256814.65	331
CVC4 (exp)	0	26138	939541.88	223
SMT-RAT	14	16329	25022167.18	10071
STP-CMSat4	0	26124	979823.26	290
STP-CMSat4 (mt-v15)	14	17486	70768.54	8914
STP-CMSat4 (v15)	16	26181	756053.05	217
STP-MiniSAT (v15)	16	25587	2265496.19	811
Yices	0	25647	2106185.67	767
[MathSat]	0	25895	1658560.27	519
[z3]	0	26108	1052484.34	306

Table 5.1: SMT-COMP 2015 results in the QF_BV division (Main Track, sequential performance)

Solver	# solved	# res. out	# unknown	time
CVC4	5043	805	0	7016.79
MathSat 5.3.6	4690	1157	1	13279.67
SMT-RAT (old)	2720	3127	1	21857.89
SMT-RAT (new)	3164	2684	0	19423.97
Yices	4660	1188	0	9913.71
z3	5105	743	0	12350.01

Table 5.2: Current results on a selection of 5847 benchmarks from the QF_BV benchmarks of SMT-COMP 2015

5.2 BVModule Evaluation

Table 5.2 shows the results of a comparison of the solvers CVC4, MathSat, Yices and z3, in their versions which participated in the SMT-COMP 2015, and the solver SMT-RAT. "SMT-RAT (old)" denotes the version of the competition, "SMT-RAT (new)" refers to the current state of the implementation. The used set of 5847 benchmarks was chosen randomly from the set of all QF_BV benchmarks of SMT-COMP 2015. All solvers were given a maximum execution time of 60 CPU seconds and a memory limit of 5 GB.

The column "# res. out" lists the number of runs of the corresponding solver which resulted in a resource exhaustion, i.e. a memory or time limit. Runs for which the solvers respond with the special response *unknown* can be found in the column "unknown". For the last column, which sums up the CPU time needed by each solver, computations resulting in a timeout are counted with the maximum permitted time (60 seconds).

It can be seen that we were able to improve the performance of SMT-RAT both in terms of the number of correctly solved benchmarks and the required CPU time. These performance gains in comparison to the SMT-COMP version are mainly the result of the optimizations described in Chapter 3.3.

On the other hand, even the new version of SMT-RAT still does not achieve the same results as its competitors. We attribute this to additional techniques that are implemented in the other solvers as preprocessing or abstraction steps.

5.3 IntBlastModule Evaluation

One of the most important configuration options of the `IntBlastModule` is the numerical setting MAX_WIDTH, which controls the maximum number of bits that are used to encode a single integer variable. Table 5.3 compares the running times and the number of solved instances for different choices of MAX_WIDTH. The settings for the benchmarks in this section comprise a CPU timeout of 1200 seconds, a wallclock timeout of 600 seconds and a memory limit of 25 GB.

Two general observations can be made: First, a higher MAX_WIDTH has great impact on the running time of the algorithm. This is not surprising, as a higher number of bits leads to a quick increase in the global number of propositional variables. Second, although an increase of MAX_WIDTH generally makes it possible to find more solutions using the bitvector solver, the figures convey the opposite relation. This is due to the fact that the solver cannot complete many of its computations in the case of a high MAX_WIDTH, such that the solutions cannot

MAX_WIDTH	# solved	# res. out	# unknown	time
2	7386	160	929	8270.01
3	7406	332	737	22132.51
4	7392	487	596	35854.48
6	7288	740	447	50526.63
8	7204	922	348	79110.69
12	6986	1265	224	133762.47

Table 5.3: Performance on different settings for MAX_WIDTH

Solver	# solved	# res. out	# unknown	time
AProVE	8266	201	8	6162.53
SMT-RAT (IB-3)	7406	332	737	22132.51
SMT-RAT (Arith)	7421	1039	15	11043.87

Table 5.4: Comparison of AProVE to IntBlast strategy (MAX_WIDTH = 3) and arithmetic strategy (linear arithmetic, virtual substitution, cylindrical algebraic decomposition)

be found in time.

Judging from the compared numbers, a good choice for MAX_WIDTH seems to be a value around 3. It is remarkable that even only small bits per integer suffice to solve the vast majority of benchmarks.

To evaluate the overall performance of the `IntBlastModule`, we compare it to AProVE [FGM+07], the winner of the SMT-COMP 2015 in the QF_NIA category, and another SMT-RAT strategy which is based completely on arithmetic techniques, namely the Simplex algorithm, Virtual Substitution and Cylindrical Algebraic Decomposition. The results are depicted in Table 5.4.

As AProVE also uses a bit-blasting technique that is closely related to the one implemented in the `IntBlastModule`, it is interesting to see that AProVE performs better regarding the number of solved instances and the CPU time. Most probably, the difference in running time can be explained from the handling of encoding width limits. While AProVE has a variable similar to MAX_WIDTH, it does not use a constant setting for this variable, but increments it iteratively. Initially, the width limit is set to 1. This approach seems to be a good choice for reducing the average execution time.

Comparing the results of the IntBlast and the arithmetic strategy of SMT-RAT, we can conclude that the two approaches are able to solve a similar number of benchmarks. However, among the unsolved benchmarks, the relation between "res. out" and "unknown" results is significantly different. The IntBlast module is well-suited for being complemented by an alternative module as backend, which is consulted in all "unknown" cases.

All in all, the results confirm that a SAT-based approach for nonlinear integer arithmetic is feasible with a very decent performance.

Chapter 6

Conclusion

In this chapter, we sum up the most important points of our thesis and give suggestions for further related research.

6.1 Summary

The first part of our thesis was concerned with the SMT theory of bitvector arithmetic. With its rich expressivity, that is tailored to the operations of state-of-the-art CPUs, SMT(QF_BV) is a well-suited modeling framework for software and hardware verification purposes. For deciding SMT(QF_BV), we implemented a bitvector module based on the approach of flattening to propositional logic.

The prevalence of bit-blasting as a solving method for bitvector logic immediately suggests the question whether bit-blasting can also serve as a basis for handling problems on other domains. The success of SAT-based solvers like AProVE demonstrates that SAT solving on nonlinear integer arithmetic is indeed feasible. We generalized this approach towards a better modularity by reducing to bitvector logic instead of directly encoding to SAT.

Several important steps of the reduction have been identified and examined for optimization potential. One such optimization is the encoding with an offset, which allows for a better handling of linear variables. Most importantly, we showed how to integrate a module for Interval Constraint Propagation that further enables us to constrain the complexity of the created bitvector problem instances.

The evaluation of our bitvector module shows that the implemented decision procedure can already decide the majority of problems. However, there is still room for improvement, especially by applying preprocessing steps prior to the actual bit-blasting. For the integer arithmetic module, the evaluation already contains promising results.

6.2 Future Work

Throughout the development of our thesis, multiple points have arisen which would be interesting for future research.

Regarding the implementation of the bitvector module, it seems that further performance

improvements rather require new mechanisms of preprocessing or simplification than minor low-level optimizations. In the past, abstraction-based approaches like [BB09] or [BKO+07] have successfully been employed, which iteratively refine approximations of the problem. It is very likely that the idea of abstractions still has a lot of potential to further improve bitvector solving.

Another aspect to be explored is the conversion from propositional formulas, as they are generated by the bitvector module, to an equisatisfiable CNF representation. Structures like And-Inverter-Graphs [BB04] or NICE dags [MV07] provide a representation of Boolean formulas that allow for an efficient transformation to CNF and may thereby be especially suited in the context of SAT solving.

For our module concerned with integer arithmetic, there are also several aspects which could be subject of further research. An important part of the algorithm, which is probably worth being investigated, is the generation of constraint and polynomial trees. At the moment, we use the rather simple method to expand the polynomials and add up the created summands. It would be interesting to see whether using factorizations of the polynomials could lead to performance gains. Even without applying factorizations, there might be a way to rewrite monomials into polynomial trees to maximize the number of shared subtrees among the encodings of different polynomials.

Our experiments indicate that bit-blasting on integer arithmetic gives best results if the maximum width for each variable is a relatively low number. In the spirit of abstraction and refinement, an iterative incrementing of the maximum encoding width might result in a better flexibility and a faster application of the SAT solver.

With the configurability of SMT-RAT, the integer blasting module can be composed in numerous ways with the remaining modules. We believe that a smart choice of the solving strategy and the application of parallel computation has the potential to unite the strengths of the different concepts. Eventually it is possible to find heuristics that allow the strategy to adapt flexibly to its inputs. It would be exciting to examine how to optimize the solving strategy to obtain a powerful solver for nonlinear integer arithmetic.

Bibliography

[BB04] Per Bjesse and Arne Boralv. DAG-aware circuit compression for formal verification. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 42–49. IEEE Computer Society, 2004.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[BD02] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 741. IEEE Computer Society, 2002.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, pages 187–201. Springer, 1996.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th annual Design Automation Conference*, pages 522–527. ACM, 1998.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–372. Springer, 2007.

[BLNM+09] Cristina Borralleras, Salvador Lucas, Rafael Navarro-Marset, Enric Rodríguez-Carbonell, and Albert Rubio. Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In *Automated Deduction–CADE-22*, pages 294–305. Springer, 2009.

[BM05] Domagoj Babić and Madanlal Musuvathi. Modular arithmetic decision procedure. *Microsoft Research Redmond, Tech. Rep. TR-2005-114*, 2005.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2010.

[Buc85] Bruno Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In *Multidimensional Systems Theory–Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232. Springer, 1985.

[CÁ11] Florian Corzilius and Erika Ábrahám. Virtual substitution for SMT-solving. In *Fundamentals of Computation Theory*, pages 360–371. Springer, 2011.

[CDW15] Sylvain Conchon, David Déharbe, and Tjark Weber. SMT-COMP 2015. `http://smtcomp.sourceforge.net/2015/index.shtml`, 2015.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan J. Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[Chu85] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1985.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing–SAT 2015*, pages 360–368. Springer, 2015.

[CMMU03] Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. Proving termination of rewriting with CiME. In *Extended Abstracts of the 6th International Workshop on Termination, WSTâĂŹ03*, pages 71–73, 2003.

[CMTU05] Evelyne Contejean, Claude Marché, Ana P. Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[Coo71] 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.

[Dav87] Ernest Davis. Constraint propagation with interval labels. *Artificial intelligence*, 32(3):281–331, 1987.

[DDM06] Bruno Dutertre and Leonardo De Moura. The YICES SMT Solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2), 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[FGM+07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. *SAT solving for termination analysis with polynomial interpretations*. Springer, 2007.

[FORS03] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonizer and Solver. In *Computer Aided Verification: 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001. Proceedings*, volume 2102, page 246. Springer, 2003.

[Fuh07] Carsten Fuhs. SAT-based Methods for Automated Termination Analysis with Polynomial Orderings. Diploma thesis, RWTH Aachen, Germany, April 2007.

[GBD02] Vijay Ganesh, Sergey Berezin, and David L. Dill. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design*, pages 171–186. Springer, 2002.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[Gom63] Ralph E. Gomory. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming*, 64:260–302, 1963.

[HBJ+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Computer Aided Verification*, pages 680–695. Springer, 2014.

[HC00] Chung-Yang Huang and Kwang-Ting Cheng. Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In *Proceedings of the 37th Annual Design Automation Conference*, pages 118–123. ACM, 2000.

[Hil00] David Hilbert. Mathematische Probleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, 1900:253–297, 1900.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view*. Springer Science & Business Media, 2008.

[Lev73] Leonid A Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[Lov78] Donald W Loveland. Automated theorem proving: A logical basis (fundamental studies in computer science), sole distributor for the usa and canada, 1978.

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *Computer Aided Verification*, pages 475–478. Springer, 2004.

[Mat70] Yuri V. Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2):279–282, 1970.

[MR98] M. Oliver Möller and Harald Rue. Solving bit-vector equations. In *Formal Methods in Computer-Aided Design*, pages 36–48. Springer, 1998.

[MV07] Panagiotis Manolios and Daron Vroon. Efficient circuit to cnf conversion. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 4–9. Springer, 2007.

[NM65] John A. Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[PICW04] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. An efficient finite-domain constraint solver for circuits. In *Proceedings of the 41st annual Design Automation Conference*, pages 212–217. ACM, 2004.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *Computer Aided Verification*, pages 500–504. Springer, 2002.

[Sch13] Stefan Schupp. Interval constraint propagation in SMT compliant decision procedures. Master's thesis, RWTH Aachen, Germany, March 2013.

[Sho82] Robert E. Shostak. Deciding combinations of theories. In *6th Conference on Automated Deduction*, pages 209–222. Springer, 1982.

[TO13] Van Khanh To and Mizuhito Ogawa. raSAT: SMT for polynomial inequality. *Research report*, 2013:1–23, 2013.

[Tse83] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[Wei98] Volker Weispfenning. *A new approach to quantifier elimination for real algebra.* Springer, 1998.