DISS. ETH NO. 19589

Termination Analysis for Bit-Vector Programs

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences (Dr. sc. ETH Zurich)

presented by

Christoph Michael Wintersteiger

Dipl.-Ing., Johannes Kepler Universität Linz, Österreich

Date of birth
May 15, 1979
citizen of Austria

accepted on the recommendation of

Prof. Dr. David Basin
Prof. Dr. Daniel Kröning
Dr. Leonardo de Moura

2011

Abstract

Recent advances in software termination analysis have shown that program termination can be decided efficiently for many practically relevant problems, despite the fact that the Halting Problem in general is undecidable. This dissertation presents a new algorithm for termination analysis, called *Compositional Termination Analysis*, which is based on compositional (or transitive) transition invariants. This algorithm depends on an underlying ranking relation synthesis engine and this dissertation presents two such engines that are able to synthesize Bit-Vector ranking relations. This class of ranking relations is especially important for verification of embedded software or for software that interacts with hardware, like device drivers.

Furthermore, a method for certification of decision procedures for quantified Boolean formulae (QBF) is presented; a requirement for one of the ranking relation synthesis methods and many other applications of QBF. Since decision procedures for QBF face performance problems in practice, an alternative and richer logic (quantified Bit-Vector logic with uninterpreted functions) is proposed. This logic enables decision procedures to be more efficient for many practically relevant formulas and at the same time it enables a more convenient and efficient translation of verification and synthesis problems to the input of the decision procedure.

All of the methods described in this dissertation are compared in a substantial experimental evaluation which clearly demonstrates the advantages in runtime or precision of the new methods over existing techniques.

Zusammenfassung

Im Gebiet der Terminierungsanalyse für Software gab es in den letzten Jahren große Fortschritte auf vielen praktisch relevanten Problemen, trotz der Unentscheidbarkeit des Halteproblems im generellen Fall. Diese Dissertation präsentiert einen neuen Algorithmus für Terminierungsanalyse, genannt *Compositional Termination Analysis*, welcher auf compositional (transitiven) Transitionsinvarianten basiert. Dieser Algorithmus hängt von einem externen Algorithmus zur Synthese von Rank-Relationen ab und zwei solche Algorithmen werden vorgestellt. Diese Algorithmen synthetisieren Rank-Relationen für Programme mit Bit-Vector Variablen, einer Klasse von Programmen die besonders im Bereich der Verifikation von embedded und hardware-nahen Software, wie etwa Gerätetreibern, von großer Bedeutung ist.

Des weiteren wird eine Methode zur Zertifizierung von Entscheidungsprozeduren für quantifizierte Boole'sche Formeln (QBF) präsentiert; eine Voraussetzung für eine der Synthesemethoden für Rank-Relationen und viele andere QBF-Anwendungen. Da Entscheidungsprozeduren für QBF in der Praxis Performanceprobleme aufweisen wird eine alternative Logik (quantifizierte Bitvektor-Logik mit uninterpretierten Funktionen) vorgeschlagen. Diese Logik erlaubt effizientere Entscheidungsprozeduren für viele praktisch relevante Probleme, während die Übersetzung von Verifikations- und Syntheseproblemen zur Eingabe der Entscheidungsprozedur vereinfacht wird.

Alle Methoden die in dieser Dissertation beschrieben werden, wurden einer umfangreichen experimentellen Evaluierung unterzogen, welche die Laufzeit- oder Präzisionsvorteile der neuen über vergleichbare existierende Techniken klar aufzeigt.

Contents

Chapter 1

Introduction

From the very beginnings of digital system design the question of whether a given system or program would always complete its calculation within a finite amount of time has been of paramount importance to the designers. A potentially non-terminating (sub-)system or program often implies undesired consequences for the system it is a part of. For example, a non-terminating subroutine in a device driver could mean that the device becomes unusable or that the whole system becomes unresponsive.

With the advent of formal algorithm analysis, it soon became clear that there is no general solution to this problem: When Alan Turing introduced the Turing machine as a formalism for programs, he asked for a set of system states to be declared as *final states* in every Turing machine. If the machine enters one of these states, all further execution is suspended, i.e., the machine *halts*. For any given Turing machine, the problem of determining whether it terminates is then simply stated as *Does the machine eventually enter a final state for all possible inputs?* In this context, the problem is then called the *Halting Problem*.

While the Halting problem has a succinct formulation, it is by no means a simple problem. In fact, the problem is *undecidable*, which means that for some programs (or Turing machines), it is impossible to find an answer to the question posed. It is for this reason that the Halting Problem has received little attention from practitioners for many years. In recent years however, it has been demonstrated that the problem does have a solution in many cases that are important in practice. The most important result, which revived this area of research, is the *Terminator* algorithm by Cook, Podelski and Rybalchenko [35]. This algorithm first enabled widespread application of termination provers to large-scale software and it was used to prove termination of Windows device driver routines, where it found many important termination bugs.

The Terminator algorithm ingeniously decomposes a given program into several smaller programs which are then proven to be terminating in isolation, followed by additional termination checks for combinations of these parts, if they are necessary. A complete decision procedure cannot exist for an undecidable problem, implying that the Terminator algorithm cannot be complete. In practice however, it turns out that these cases are rare, making the Terminator algorithm a *practical* solution to the Halting Problem.

This dissertation presents a new algorithm, dubbed *Compositional Termination Analysis*, that is inspired by the Terminator algorithm. It is based on the observation that intermediate termination arguments often describe transitive (or *compositional*) behavior, which allows for an immediate generalization of the termination argument to the whole program, removing the need for further checks of ever more intricate combinations of program parts. By recognizing such cases, Compositional Termination Analysis is able reach a conclusive result in much less time than the Terminator algorithm requires.

Just like the Terminator algorithm, Compositional Termination Analysis depends on a ranking relation synthesis engine which synthesizes termination arguments for the intermediate program parts. While this problem is undecidable in general too, it is decidable for some classes of programs. This is the case for finite-state programs, since all potentially suitable ranking relations could, in theory, be enumerated. The class of finite-state Bit-vector programs is particularly important in the context of embedded software, where strict hardware limits have to be respected by the software, especially in terms of memory consumption. Also, termination of many infinite-state programs depends only on a small part of the program which is often representable by a finite-state program. Despite the need for termination proofs in embedded software and operating systems, no bit-precise solutions that faithfully model over- and underflows in addition and other operators have existed until now. This dissertation presents two ranking relation synthesis methods for finite-state programs. Each of them is able to synthesize ranking relations which are composed of linear functions over Bit-vector variables; the underlying synthesis methods however are fundamentally different: while one of them relies on a translation of the problem to an equivalent Integer Linear Programming (ILP) problem, the other creates an equisatisfiable quantified Boolean formula (QBF).

The Terminator algorithm and Compositional Termination Analysis not only require an answer to the question of whether a termination argument (a suitable ranking relation) exists for some intermediate program. They also require concrete ranking relations, i.e., not only do they require an answer, but also a solution. This is trivial in the case of ILP, but highly non-trivial in the case of QBF. No generally accepted format for QBF solutions exists. This dissertation contains a proposal for such a format, which has not only proven to be useful in practice, but has also uncovered many issues that inhibit efficient certification of modern QBF decision procedures.

QBF decision procedures face performance problems in practice, but the logic itself does have a huge number of applications. Motivated by this fact, a related logic, namely quantified Bit-vector logic, is investigated in this dissertation. Experiments on interesting problems (including ranking relation synthesis) indicate that a relatively simple decision procedure, which is capable of producing solutions in a trivial manner, is up to five orders of magnitude faster than modern QBF decision procedures. Quantified Bit-vector logic not only constitutes a very attractive formalism for a vast number of (low-level) software and hardware verification problems, but it appears to allow more practical and efficient decision procedures than QBF.

All of the methods presented in this dissertation are supported by a substantial experimental evaluation. Not only does this evaluation provide a strong argument for the superiority of the new methods over existing methods in use, it also makes a case for the feasibility of bit-precise termination proofs in the software design process. Through the increase in performance obtained by the methods presented here, checking software for termination before it is released becomes possible and will, hopefully, become common practice.

Organization. The chapters of this dissertation are organized as follows: Chapter 2 introduces definitions that subsequent formalizations depend on and discusses related work that is of interest in the general context of modern termination proving. Chapter 3 introduces Compositional Termination Analysis and discusses its relation to the Terminator algorithm. Chapter 4 presents two methods for ranking relation synthesis for Bit-vector programs. One of these methods requires a certifying solver for the validity problem of quantified Boolean formulae. Chapter 5 proposes a technique for certification that may be used for this purpose. In practice however, the resulting ranking relation method suffers from performance problems. A remedy for this problem is a decision procedure for the satisfiability problem of a related logic presented in Chapter 6. This decision procedure is, in practice, multiple orders of magnitude faster than previous approaches and may present an alternative to other methods based on decision procedures for quantified Boolean formulae. Chapter 7 presents an experimental evaluation of all the methods described in this dissertation. Each method is evaluated thoroughly on large sets of benchmarks, sometimes in multiple configurations, and compared to state-of-the-art methods from the respective areas. Finally, Chapter 8 concludes and provides an outlook on future research.

Chapter 2

Background

The methods described in later chapters of this dissertation rely on some fundamental definitions and related work which are covered in this chapter. The first section of this chapter presents preliminaries on the SAT and QBF problems, which are commonly used for bit-precise analysis of programs in general. The second part of this chapter focusses on termination analysis, providing preliminaries and definitions of terms used in connection with termination analysis.

2.1 The SAT and QBF Problems

In this dissertation the propositional satisfiability problem (SAT) and the decision problem for quantified Boolean formulae, commonly abbreviated 'QBF', are frequently referred to. Before these problems can be defined, some preliminary definitions are required:

The set of Boolean values is $\mathbb{B}$ and it is equal to $\{\top, \bot\}$, where $\top$ represents truth and $\bot$ represents untruth. The usual Boolean algebra $(\mathbb{B}, \wedge, \vee, \neg)$ is required.

Definition 1 (Literal)**.** *A literal represents either a variable or its negation, denoted* v *or* $\neg v,$ *for* $v \in \mathbb{B}$*.*

An infinite set V of Boolean variables is assumed. The set of literals contains all of these variables and their negations $\neg v$ with $v \in V$. As usual, the notion of negation to is extended to literals and $\neg\neg v$ is identified with v.

Definition 2 (Clause)**.** *A* clause *is a disjunction of literals and is denoted as* $\{l_1, l_2, \dots\}$ *for literals* $l_1, l_2, \dots$*.*

Note that the set-theoretic notation for clauses (intentionally) implies that literals are assumed to be unique within clauses, i.e., the number of occurrences of a given literal within a clause is either 0 or 1.

A clause is tautological if it contains a literal and its negation (and therefore semantically equivalent to $\top$). An empty clause is a clause without literals (semantically equivalent to $\bot$).

Definition 3 (Matrix)**.** *A conjunction of clauses is called* matrix*.*

Note that a matrix thus takes the shape $\bigwedge(\bigvee l_{i,j})$ for some literals $l_{i,j}$ and that it is therefore in *conjunctive normal form (CNF)*.

Definition 4 (Assignment)**.** *A mapping* $\alpha : V \to \mathbb{B}$ *that maps* all *variables that occur in a matrix to values from* $\mathbb{B}$ *is an* assignment*.*

Should the mapping be only partial, i.e., contain only mappings for *some* of the variables of a matrix, it is called a *partial assignment*. For brevity, the replacement of all variables in a matrix ϕ with their mapped value of some assignment α is denoted as ϕ_α.

Deciding whether a given matrix has a satisfying assignment is the well-known satisfiability or SAT problem:

Definition 5 (Satisfiability (SAT) Problem)**.** *Let ϕ be a matrix. The* SAT problem *is to decide whether there exists an assignment α to the variables in ϕ such that $\phi_\alpha \equiv \top$.*

Alternatively, the same problem may be stated as deciding the truth of a formula where all of the variables are quantified existentially, i.e., as $\exists V \,.\, \phi$.

Allowing universal as well as existential quantification of the variables results in quantified formulas:

Definition 6 (QBF)**.** *A quantified Boolean formula (QBF) is a concatenation of a quantifier prefix and a matrix of clauses.*[1]

For the purposes of this dissertation it is enough to consider closed QBFs in prenex normal form, i.e., every variable is bound by a quantifier and quantifiers appear only in one (linear) quantifier prefix.

In order to define the semantics of a QBF, let the function $\Omega\colon V \to \{\exists, \forall\}$ be a function that marks variables either as existential ($\exists$) or as universal ($\forall$).

Definition 7 ($[\![\cdot]\!]$, Semantics of a QBF)**.** *The semantics $[\![\phi]\!]$ of a QBF ϕ is defined recursively by expanding the outermost variable v of ϕ as follows. If $\Omega(v) = \exists$ then define $[\![\phi]\!]$ as $[\![\phi\{v \leftarrow \bot\}]\!] \vee [\![\phi\{v \leftarrow \top\}]\!]$, where the cofactor $\phi\{v \leftarrow c\}$ is ϕ in which every occurrence of v is replaced by the Boolean constant c. If on the other hand, $\Omega(v) = \forall$, then $[\![\phi]\!] = [\![\phi\{v \leftarrow \bot\}]\!] \wedge [\![\phi\{v \leftarrow \top\}]\!]$.*

More precisely, for $c = \bot$, clauses containing $\neg v$ are deleted and v is removed from all clauses, and similarly for $c = \top$. Note that empty clauses (respectively empty QBFs) are equivalent to the Boolean constant $\bot$ (respectively $\top$) as defined above.

The validity of QBFs is often of importance; the corresponding decision problem is defined as follows:

[1] In the literature, this type of formula is sometimes referred to as a quantified SAT or QSAT formula.

Definition 8 (QBF Validity Problem)**.** *The* QBF validity problem *is to decide whether* $[\![\phi]\!] = \top$ *for arbitrary QBFs* ϕ*.*

It is helpful for our investigation to define an ordering of the variables in a QBF according to the order of the variables in the quantifier prefix:

Definition 9 (QBF Variable Ordering)**.** *Let* $v_1, v_2 \in V$*. The ordering* $<$ *is defined such that* $v_1 < v_2$ *iff* v_2 *is in the scope of* v_1*, i.e., 'larger' variables appear* after *'smaller' ones in the quantifier prefix of a QBF.*

The function Ω and the ordering $<$ are extended to literals in the natural way, i.e., $\Omega(\neg v) = \Omega(v)$, while leaving the ordering of the two literals of a variable undefined. Naturally, a variable is called innermost (resp. the outermost) if it is maximal (resp. minimal) among all variables of the QBF with respect to the order $<$.

2.1.1 Q-resolution

Kleine Büning et al. [64] describe a resolution-like calculus to determine QBF validity. To this end, they define a new operation called Q-resolution, which is a slight variation of propositional resolution that makes use of the fact that literals over universally quantified variables may be removed from a clause if no larger existential literals appear in the clause:

Lemma 1 (Forall-reduction [64])**.** *Let* C *be an arbitrary clause, and let* $\Omega(l) = \forall$ *for some* $l \in C$*. If* $\forall l' \in (C \setminus l) \,.\, l' < l$*, then* $C \equiv C \setminus l$*.*

Therefore, if a clause C contains a universal literal l that is larger than all existential literals in C, it can be removed from C. The process of removing literals according to this rule is called *forall-reduction* [14, 64]. And it can of course be applied repeatedly:

Definition 10 (Forall-reduct)**.** *The result obtained from repeated application of forall-reduction to a clause* C *until no more literals can be removed is called the* forall-reduct *of* C*.*

Using forall-reduction, Q-resolution can be defined as follows:

Definition 11 (Q-resolution [64])**.** *Two non-tautological clauses C and D can be* resolved *iff there exists a literal l such that $l \in C$ and $\neg l \in D$. The result of Q-resolution is the forall-reduct of $(C \cup D) \setminus \{l, \neg l\}$, which is called a* Q-resolvent *(or a* Q-resolvent clause*).*

Exhaustive generation of all Q-resolvents results in a sound and complete algorithm for the QBF validity problem [64]. The Q-resolution rule may also be coupled with a complete search algorithm, which is the basis for many QBF solving algorithms usually called *QBF-solvers* (e.g. [12, 14, 50]).

2.1.2 Satisfiability Modulo Theories

In this dissertation, frequent reference is made to so-called Satisfiability Modulo Theories (SMT) solvers or the SMT Library. These solvers are SAT decision procedures which are extended to handle richer logics by the use of (sometimes external) theory decision procedures. SMT solvers find a solution to the propositional skeleton of a formula, leaving only a set of theory-specific formula atoms to be decided by the external theory solver.

The SMT Library is a collection of logics defined over a small set of background theories. With each of those logics a set of benchmarks is associated and an annual competition (SMT-COMP) is held to determine the state of the art in solving these problems. In the context of this dissertation, the theory of quantifier-free Bit-vectors, or SMT QF_BV, is of importance. This theory allows formulas over variables that range over Bit-vectors of some fixed size and gives an interpretation to the usual Bit-vector operations like addition and multiplication (with overflow), and the bit-wise logical operations. A precise definition and the associated benchmarks is available at the web-site of the SMT Initiative at `http://www.smtlib.org/`.

2.2 Termination

2.2.1 Preliminaries

In this section notation for programs and their basic properties are defined. Programs are modeled as *transition systems*, abstracting from the many different types of instructions in actual programming languages.

Definition 12 (Transition System). *A transition system (program) P is a three tuple $\langle S, I, R\rangle$, where*

- *S is a (possibly infinite) set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is the transition relation.*

Were one to execute such a program, it would start in one initial state $s_0 \in I$. Thereafter one or more transitions may become enabled in R, i.e., $(s_0, s_1) \in R$ for some $s_1 \in S$. This represents a transition of the system from state s_0 to state s_1 (for nondeterministic systems, there may be multiple s_1). The following definition captures this more formally.

Definition 13 (Computation). *A* computation *of a transition system is a sequence of states $s_0, s_1, \ldots$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.*

Note that a computation is not required to be finite. If a program allows infinite computations, it is clearly not terminating as there is at least one computation which continues indefinitely. Therefore, termination of a program is defined as follows.

Definition 14 (Termination). *A program is terminating if and only if all its computations are finite.*

To analyze programs it is necessary to reason about the behavior of the program in general. This means that it is sometimes not enough to reason about the transition relation by itself. Instead, a means of reasoning about multiple steps or even an infinite number of steps is required. This is achieved through the notion of the transitive closure of a relation.

Definition 15 (Transitive Closure)**.** *Let $R : X \times X$ be a binary relation. The* transitive closure *R^+ of R is defined as*

$$R^+ = \bigcup_{i \in \mathbb{N}} R^i \ ,$$

where $R^0 = R$ and

$$R^i = R^{i-1} \cup \{(x_1, x_2) | \exists x_3 \in X \ . \ (x_1, x_3) \in R^{i-1} \wedge (x_3, x_2) \in R^{i-1})\} \ .$$

Intuitively, R^+ is the set of all possible multi-step transitions of the system, e.g., if a program allows only the computation $s_0, s_1, s_2,$ then $R = \{(s_0, s_1), (s_1, s_2)\}$ and $R^+ = \{(s_0, s_1), (s_1, s_2), (s_0, s_2)\}$.

Additionally, a variation on the transitive closure which is also reflexive is required:

Definition 16 (Reflexive Transitive Closure)**.** *The* reflexive transitive closure *R^* of a binary relation $R : X \times X$ is defined as*

$$R^* = R^+ \cup \{(x, x) | x \in X\} \ .$$

These definitions enable precise definition of the states that a program is able to reach.

Definition 17 (Reachable States)**.** *The set of reachable states of a program $\langle S, I, R\rangle$ is $R^*(I) = \{s | \exists i \in I \ . \ (i, s) \in R^*\}$.*

Correspondingly, the set of reachable transitions is defined as follows.

Definition 18 (Reachable Transitions)**.** *The set of reachable transitions of a program $\langle S, I, R\rangle$ is defined as*

$$R_I = \{(s_1, s_2) \mid (s_1, s_2) \in R \wedge \exists s_0 \in I \ . \ (s_0, s_1) \in R^*\} \ .$$

2.2.2 Proving Termination

A straightforward approach to proving termination is to construct a so-called *ranking function*, an idea which was suggested by Turing in 1949:

> Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. [99]

The continually decreasing quantity is a function that has as its domain the state space of the program S (a configuration of a Turing machine) and as its range, the *rank*, the natural numbers. The signature of this function therefore is $f : S \to \mathbb{N}$. The important requirement on this function is that its value decreases with each step of the program (Turing machine) in every computation. If it is possible to construct a function that fulfils this requirement, the program clearly is terminating, as the rank can never decrease beyond 0.

This intuitive idea about termination proving is formalized via the notion of *well-foundedness* of relations, a concept due Marimanoff [73]:

Definition 19 (Well-foundedness)**.** *A binary relation $R : X \times X$ is* well-founded (wf.) *iff every non-empty subset of X has a minimal element m with respect to R, i.e.,*

$$\forall S \subseteq \mathcal{P}(X) \; . \; S \neq \emptyset \Rightarrow \exists m \in S \; . \; \forall s \in S \; . \; (s, m) \notin R \; ,$$

where $\mathcal{P}(X)$ is the power set of X.

Well-foundedness ensures that a relation does not allow any infinitely descending chains. This maps directly onto infinite computations of programs:

Lemma 2. *If the transition relation of a program is well-founded, the program is terminating.*

Proof. Every computation $s_0, s_1, s_2, \ldots$ represents a series of sets of states $S_0 = \{s_0\}$, $S_1 = \{s_0, s_1\}$, $S_2 = \{s_0, s_1, s_2\}$, $\ldots$. Since R is well-founded, each of those sets has a least element $m_i \in S_i$ with respect to R. In the case of S_0, the least element is trivially s_0. Since $(s_0, s_1) \in R$, s_1 can not be the least element of S_1. This applies to all further sets as well, s.t., s_0 must be the least element of the computation. The same argument applies recursively from s_1. $\square$

As is apparent from the definition of well-foundedness, any sub-relations $R' \subseteq R$ of a well-founded relation R are also well-founded. To prove termination of a program with transition relation R, it is therefore enough to find a well-founded relation T that is a superset of R.

Corollary 1. *Let $P = \langle S, I, R\rangle$ and $R \subseteq T$. If T is well-founded, P terminates.*

In practice, it is often easier to construct a well-founded T that covers R than to directly prove R well-founded. Since T is also a binary relation and it is usually constructed with the help of ranking functions it is called a *ranking relation*. It is thus Turing's suggestion to construct a well-founded ranking relation to prove program termination. In the rest of this dissertation, this approach is therefore referred to as *Turing's method*:

Theorem 1 (Turing's method)**.** *The program $\langle S, I, R\rangle$ terminates iff*

$$\exists T \,.\, T \supseteq R_I \wedge T \textit{ is well-founded} \,.$$

Note that the implication indeed holds in both directions, as a well-founded T does not exists for (reachable parts of) transition relations that allow infinitely descending chains.

Since it is generally considered hard to construct a suitable well-founded relation for Turing's method, Podelski and Rybalchenko propose a different approach based on a slightly weaker notion than well-foundedness:

Definition 20 (Disjunctive well-foundedness [82]). *A binary relation $T : X \times X$ is* disjunctively well-founded (d.wf.) *if it is a finite union of well-founded relations, i.e., $T = \bigcup_{i=0}^{n} T_i$.*

A union of well-founded relations is not, of course, well-founded in general. To prove termination of a program $\langle S, I, R \rangle$ it is therefore not enough to show that there exists a disjunctively well-founded T that covers R_I. Instead, a slightly different notion is required:

Definition 21 (Transition Invariant [82]). *A* transition invariant T *for program $P = \langle S, I, R \rangle$ is a superset of the transitive closure of the reachable transitions, i.e., $R_I^+ \subseteq T$.*

Podelski and Rybalchenko then show that disjunctive well-foundedness of a transition invariant is equivalent to program termination. The distinctive feature of this method is the use of disjunctive well-foundedness; it is therefore referred to as the *disjunctive method* in this dissertation:

Theorem 2 (Disjunctive Method [82]). *A program $P = \langle S, I, R \rangle$ is terminating iff there exists a d.wf. transition invariant for P, i.e.,*

$$\exists T \,.\, T \supseteq R_I^+ \wedge T \textit{ is disjunctively well-founded}\,.$$

When comparing Turing's to the disjunctive method, the trade-off between the two becomes apparent: Turing's method requires a T which covers (in the worst case) R, while the disjunctive method requires a T which covers (in the worst case) R^+, where the latter is usually much larger than R. Should it be required, the inclusion is also harder to verify in an automated fashion. On the other hand, Turing's method requires T to be well-founded, while the disjunctive method requires only disjunctive well-foundedness, which is usually easier to establish by construction.

An algorithm that uses the disjunctive method to its advantage is the *Terminator* algorithm, proposed by Cook et al. [34,35] and depicted in Algorithm 1.

```
input : Program $\langle S, I, R \rangle$
1  $T := \emptyset$
2  while True do
       /* Binary Reachability Analysis */
3      if $R_I^+ \subseteq T$ then
4          return terminating
5      else
6          $\rho :=$ a binary relation s.t. $\rho \subseteq R_I^+$ but $\rho \not\subseteq T$
7      end
       /* Ranking Relation Synthesis */
8      if $\rho$ is not well-founded then
9          return non-terminating
10     else
11         $W :=$ a ranking relation, i.e., $\rho \subseteq W$ and $W$ wf.
12         $T := T \cup W$
13     end
14 end
```

Algorithm 1: The Terminator Algorithm [35].

This algorithm iteratively constructs a disjunctively well-founded termination argument T and to that end, it requires two crucial components: A means to check the inclusion of the transitive closure of R in T (at line 3) and to construct a ranking relation W (at line 11).

The first of these components is (in practice) implemented using a *Reachability* Checker (a form of *Model Checker*[2]). The reachability checker takes as input a program $\langle S, I, R \rangle$ and a set of 'safe' states $\pi \in \mathcal{P}(S)$. It then proceeds to check whether there exists a computation of the program that reaches a state that is *not* in π, i.e., an 'unsafe' state. If this is the case, it returns the corresponding computation as a proof of violation of the property. This computation is then called a *counterexample*.

[2]For an introduction to Model Checking see [25].

Definition 22 (Counterexample). *A counterexample for program $\langle S, I, R\rangle$ with respect to a safety property $\pi \in \mathcal{P}(S)$ is a computation $s_0, s_1, \ldots, s_n$ where $s_0 \in I$, each $(s_i, s_{i+1}) \in R$, and $s_n \notin \pi$.*

A counterexample exists whenever $R^*(I) \not\subseteq \pi$. For the Terminator algorithm however, a reachability checker is not applicable, because T is a binary relation. Therefore, Cook et al. propose the concept of *Binary Reachability Analysis* [35]. In this method, the input is a program and a binary relation T that serves as the property. The Binary Reachability Analyzer then searches for a computation of the program which contains a transition not contained in T. More precisely, a computation $s_0, s_1, \ldots, s_n$ is a counterexample in Binary Reachability Analysis iff $(s_i, s_n) \notin T$ for some $i < n$.[3] In practice, Binary Reachability Analysis is usually implemented using a traditional reachability checker, which is possible through instrumentation of the input program. This transformation is explained in great detail by Cook et al. [35].

The Terminator algorithm iteratively constructs a termination argument T with the help of Binary Reachability Analysis. Initially, an empty termination argument is used, i.e., $T_0 = \emptyset$. Then, Binary Reachability Analysis extracts a counterexample, i.e., it finds a computation of the program which is not contained in T. From this counterexample, a binary relation ρ containing all transitions that occur in the counterexample is obtained, i.e., $\rho = \{(s_0, s_1), \ldots, (s_{n-1}, s_n)\}$, which is a subset of R_I^+ and not a subset of T, as required by line 6 of Algorithm 1.

If there is no such counterexample, termination is proven as the requirements for application of Theorem 2 are fulfilled. Otherwise, a well-founded ranking relation W that includes ρ is constructed. Finally, the current termination argument is updated disjunctively, i.e., $T_{i+1} = T_i \cup W$, preserving disjunctive well-foundedness of T. The process is then repeated until no more counterexamples are found.

[3]Note that the literature sometimes (erroneously) refers to the Terminator algorithm as Binary Reachability Analysis.

The Terminator algorithm has been put to the test in various tools, most notably in TERMINATOR [35], ARMC [83], and in a termination prover developed for the experiments conducted for this dissertation.

2.2.3 Ranking Relation Synthesis

Automation of both methods for termination proving discussed in the previous section require an algorithm that constructs ranking relations (T). While no general solution exists to this problem (due to the undecidability of the halting problem), some solutions have been proposed for decidable subclasses of this problem.

In the following chapters of this dissertation, deeper knowledge is required about a method for complete synthesis of ranking functions (and relations) for *linear* programs; a proposal by Podelski and Rybalchenko [82]. In their setting, ranking functions are generated for transition relations of the form $R \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ which are described by systems of linear inequalities:

$$R(x, x') \;\equiv\; Ax + A'x' \leq b \qquad (A, A' \in \mathbb{Q}^{k\times n}, b \in \mathbb{Q}^k) \;,$$

where $x, x' \in \mathbb{Q}^n$ range over vectors of rationals. Other transition relations have to be encoded into such systems, which, in practice, involves an approximation of program behavior. The ranking functions derived are linear and have the codomain $\mathbb{Q}^+ = \{z \in \mathbb{Q} \mid z \geq 0\}$, which is ordered by $y \prec z \equiv y + \delta \leq z$ for some rational $\delta > 0$. Ranking functions $r : \mathbb{Q}^n \to \mathbb{Q}^+$ are represented as $r(x) = kx + c$, with $r \in \mathbb{Q}^n$ a row vector and $c \in \mathbb{Q}$. Such a function r is a ranking function with the domain $(\mathbb{Q}^+, \prec)$ if and only if the following condition holds:

$$\forall x, x' \in \mathbb{Q}^n : R(x, x') \Rightarrow \quad \begin{aligned} & kx + c \geq 0 \;\wedge \\ & kx' + c \geq 0 \;\wedge \\ & kx' + \delta \leq kx \;, \end{aligned} \tag{2.1}$$

where the first two conjuncts of the right hand side of the implication encode that the function must be bounded from below and decreasing.

Coefficients k for which equation 2.1 is satisfied can be constructed using Farkas' lemma, of which the 'affine' form given in [89] is appropriate).

Lemma 3 (Farkas' lemma)**.** *Suppose $A \in \mathbb{Q}^{n\times k}$ is a matrix, $b \in \mathbb{Q}^n$ a vector such that the system $Ax \leq b$ of inequalities is satisfiable, $c \in \mathbb{Q}^k$ is a (row) vector, and $\delta \in \mathbb{Q}$ is a rational. Then*

$$\{x \in \mathbb{Q}^k : Ax \leq b\} \subseteq \{x \in \mathbb{Q}^k : cx \leq \delta\} \tag{2.2}$$

if and only if there is a non-negative (row) vector $\gamma \in \mathbb{Q}^n$ such that $\gamma A = c$ and $\gamma b \leq \delta$.

Using this lemma, a necessary and sufficient criterion for the existence of linear ranking functions can be formulated:

Theorem 3 (Existence of linear ranking functions [82])**.** *Suppose that the transition relation $R(x, x') \equiv Ax + A'x' \leq b$ is satisfiable. R has a linear ranking function $r(x) = kx + c$ iff there are non-negative vectors $\lambda_1, \lambda_2 \in \mathbb{Q}^k$ s.t.:*

$$\lambda_1 A' = 0, \quad (\lambda_1 - \lambda_2)A = 0, \quad \lambda_2(A + A') = 0, \quad \lambda_2 b < 0.$$

In this case, m can be chosen as $\lambda_2 A'x + (\lambda_1 - \lambda_2)b$.

This criterion for the existence of linear ranking functions is necessary and sufficient for linear inequalities on the rationals, but only sufficient over the integers: there are relations $R(x, x') \equiv Ax + A'x' \leq b$ for which linear ranking functions exist, but the criterion fails, e.g.:

$$R(x, x') \;\equiv\; x \in [0, 4] \wedge x' \geq 0.2x + 0.9 \wedge x' \leq 0.2x + 1.1 \;.$$

Restricting x and x' to the integers, this is equivalent to $x = 0 \wedge x' = 1$ and can be ranked by $r(x) = -x + 1$. Over the rationals, the program defined by the inequalities does not terminate, which implies that no ranking function exists and the criterion of Theorem 3 fails.

2.2.4 Rewriting Systems

Over the past decades, considerable research effort went into analysis and construction of *reduction* or *rewriting systems.* Such systems present an alternative formalism in which the results discussed in this dissertation may be stated in. This section discusses the general subject of termination analysis of rewriting systems and relates it to the definitions given earlier as well as an overview of the relationship between the two formalisms and their respective properties. To this end, a definition of a specific class of rewriting systems over terms is required:

Definition 23 (TRS)**.** *A* Term Rewriting System (TRS) *is a pair* $(A, \rightarrow)$ *where* A *is the set of terms over some signature* Σ *and variables* V*, and the* rewrite relation $\rightarrow$ *is a binary relation on* A*, i.e.,* $\rightarrow \subseteq A \times A$*.*

As is common practice, $(x, y) \in\rightarrow$ is abbreviated as $x \rightarrow y$. A term $x \in A$ is called *reducible* iff there is a term $y \in A$ such that $x \rightarrow y$ and x is called *irreducible* (or in *normal form*) iff it is not reducible. A term $y \in A$ is called a *successor* of another term $x \in A$ iff $x \overset{+}{\rightarrow} y$, where $\overset{+}{\rightarrow}$ is the transitive symmetric closure of $\rightarrow$, i.e., when there exists a chain of intermediate terms such that $x \rightarrow \ldots \rightarrow y$. The *reflexive* transitive symmetric closure of $\rightarrow$ is denoted as $\overset{*}{\rightarrow}$ and defined as $\overset{+}{\rightarrow} \cup \{(x, x) \mid x \in A\}$. Two terms $x, y \in A$ are called *joinable*, denoted as $x \downarrow y$ iff there exists a third term $z \in A$ such that $x \overset{*}{\rightarrow} z \wedge y \overset{*}{\rightarrow} z$.

Many interesting problems can be stated in the form of a TRS of which some property is to be established. Typical properties include confluence and termination, defined as follows:

Definition 24 (Confluence)**.** *A TRS* $(A, \rightarrow)$ *is* confluent *iff for all* $x, y, z \in A$ *we have that* $x \overset{*}{\rightarrow} y \wedge x \overset{*}{\rightarrow} z$ *implies* $y \downarrow z$*.*

Intuitively, a confluent TRS ensures that no matter how paths diverge from some term x, they join at some common successor. Confluence ensures that every element of A has at most one normal form, an often desired property. However, establishing confluence is in general undecidable, in the simplest case because the relation $\rightarrow$ may allow infinite chains of applications which makes it impossible to decide whether two elements have a common successor. Rewriting systems which do not allow such chains are called *terminating:*

Definition 25 (TRS Termination). *A TRS* $(A, \rightarrow)$ *is* terminating *iff there is no infinite (descending) chain* $x \in A \rightarrow y \in A \rightarrow \ldots$

A TRS is called *convergent* if it is both, confluent and terminating.

Clearly, the notion of a terminating TRS is very similar to that of a terminating program as defined in Section 2.2. Indeed, the two are equivalent since Turing machines (deterministic and non-deterministic) can be encoded as term rewriting systems and vice versa. A Turing machine can be encoded as a TRS by encoding configurations of the machine as terms over variables representing the cells of the tape and the control state of the machine and by defining an according reduction relation that modifies a given term to represent the configuration after an initialization step, right move, or left move of the machine has taken place. In this fashion, it is possible to encode every Turing machine M into a corresponding TRS R_M such that R_M is terminating iff M is terminating.[4] Consequently, term rewriting systems may be chosen as an alternative basis for designing termination proving procedures.

Most techniques for proving termination of term rewriting systems are based on the same principle as Turing's method, i.e., by finding a well-founded ordering $\succ$ of the program states. In the terminology of rewriting systems, this means that $\succ \subseteq A \times A$ is a relation over terms. A popular choice is to assign a natural number $\phi(t)$ to every term $t \in A$ such that the natural order $<$ can be employed to obtain a ranking relation. The function ϕ is sometimes called a *measure function* which is of course synonymous with a *ranking function*. In general, terminating infinite state systems may require a well-founded ordering not representable by a mapping into natural numbers. Ordinal numbers,

[4]See Chapter 5 in [3] for a precise definition of this transformation.

as suggested by Turing [99], may be a remedy. However, natural numbers are sufficient for finitely branching rewriting systems, i.e., for systems where every term has only a finite number of successors (cf. Lemma 2.3.3 in [3]). This is especially significant in the context of this dissertation as the focus is on bit-vector programs, for which a corresponding TRS is always finitely branching. Dershowitz [39] provides a survey of the fundamental techniques of termination proving for rewrite systems.

Chapter 3

Compositional Termination Analysis

This chapter presents a new method for proving termination of programs called *Compositional Termination Analysis*. It is based on the *Terminator algorithm* by Cook, Podelski and Rybalchenko [35]. The distinctive features of Compositional Termination Analysis are 1) the use of compositional (transitive) transition invariants and 2) the use of a bounded analysis of the state space of programs. Both of these features allow for an increase in performance as exhaustive analysis of intermediate relations or programs is often considerably less demanding than in the Terminator algorithm.

3.1 Motivation

Substantial progress towards the applicability of procedures that compute termination arguments for industrial code was achieved by the Terminator algorithm, proposed by Cook, Podelski, and Rybalchenko [35]. Their approach combines detection of ranking relations for program paths with Binary Reachability Analysis. The key concept of the algorithm is to encode an intermediate termination argument into a program annotated with a (binary) assertion, which is then passed to a reachability checker. Any counterexample for the assertion produced by the reachability checker contains a path that violates the intermediate termination argument. The counterexample path is then used to compute a better termination argument with the help of methods that synthesize ranking relations for program paths.

Experiments with different implementations have shown that the bottleneck of this approach is Binary Reachability Analysis [33, 35]: Cook et al. report more than 30 hours of runtime for some of their benchmarks, while the time for synthesizing ranking relations for a given program path makes up less than 1% of the runtime [35], i.e., it is insignificant in comparison. This problem unfortunately applies to both instances of finding a counterexample to an intermediate termination argument and to proving that no such counterexample exists. Part of the reason for the difficulty of the safety checks is their dual role: they ensure that a disjunctively composed termination argument is correct and they need to provide sufficiently deep counterexamples for the generation of further ranking relations.

Example 1. *Consider the following trivial example:*

```
int i=0;
while i<255 do
    i++;
end
```

where i may be considered either a bounded or an unbounded integer.

To prove termination of this program a d.wf. transition invariant for the loop must be constructed. Binary Reachability Analysis very quickly finds a potentially non-terminating path, i.e., a path not included in the (initially) empty termination argument. Usually, the first counter-example it finds executes the *loop once. Modern ranking function synthesis tools (e.g., [81]) are able to find a ranking function along the path of this counter-example in a negligible amount of time.*

Now, consider the same example as a part of a large program. In this case, computation of a path to the beginning of the loop may already exceed the computational resources available. In the terminator algorithm, termination is also proven by showing absence of counterexamples through Binary Reachability Analysis. The consequence is that for complex loops containing many (non-deterministic) branches, the underlying reachability checker is confronted with a very hard problem.

This chapter proposes a new algorithm for termination analysis called *Compositional Termination Analysis (CTA)* which addresses the issues identified with the Terminator algorithm as follows: 1) A light-weight criterion for termination based on *compositionality* of transition invariants is used. 2) Instead of using full counterexample paths, the algorithm applies the path ranking procedure directly to increasingly deeper unwindings of the program until a suitable ranking argument is found.

Furthermore, soundness of CTA is proven and (relative) completeness is discussed. The techniques presented here were initially published in [67].

3.2 Theory

Much of the theory in this section depends on the relational composition operator $\circ$ for two relations $A, B : X \times X$. It is defined as

$$A \circ B := \{(s, s') \,|\, \exists s''.(s, s'') \in A \wedge (s'', s') \in B\} \ .$$

Note that a relation R is *transitive* if it is closed under relational composition, i.e., when $R \circ R \subseteq R$. To simplify presentation, R^i is defined to represent repeated application of the relation compositional operator to R and itself, i.e., $R^1 := R$ and $R^n := R \circ R^{n-1}$ for any relation $R : X \times X$.

Just like in the Terminator algorithm, disjunctively well-founded transition invariants are the core of the algorithm proposed here. However, a trivial subclass is especially important:

Definition 26 (Compositional Transition Invariant)**.** *A d.wf. transition invariant T is called* compositional *if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.*[1]

A compositional transition invariant is also well-founded (not only disjunctively well-founded), since it is an inductive transition invariant for itself (cf. Corollary 1 in [82]). Using this observation and Theorem 2, implies the following corollary:

Corollary 2. *A program P terminates if there exists a compositional transition invariant for P.*

In Binary Reachability Analysis, the reachability checker needs to compute a counterexample to an intermediate termination argument, which is often difficult. The counterexample begins with a *stem*, i.e., a path to the entry point of the loop. For many programs, the existence of a d.wf. transition invariant does not actually depend on this entry state. For example, termination of the trivial loop in Example 1 does not actually depend on the initial value of i, nor does it depend on the concrete upper limit 255. The assurance of progress towards some upper limit is enough to conclude termination.

The other purpose of the reachability checker in the Terminator algorithm is to check that a candidate transition invariant indeed includes R_I^+. To this end, note that the (non-reflexive) transitive closure of R is essentially an unwinding of program loops:

$$R^+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{i=1}^{\infty} R^i \ .$$

[1] The term *compositional* is used instead of *transitive* for transition invariants in order to comply with the terminology in the existing literature [82]. In the remainder of this dissertation, relations are called *transitive* and transition invariants are called *compositional*.

Instead of searching for a d.wf. transition invariant that is a superset of R^+, the problem can therefore be decomposed into a series of smaller ones. Consider a series of loop-free programs in which R is unwound k times, i.e., the program that contains the transitions in $R^1 \cup \ldots \cup R^k$.

Lemma 4. *Let $P = \langle S, I, R\rangle$ and $k \geq 1$. If there is a d.wf. relation T_k with $\bigcup_{i=1}^{k} R^i \subseteq T_k$ and T_k is also transitive, then T_k is a compositional transition invariant for P.*

Proof. The goal is to show that T_k is a transition invariant for P, i.e., $R_I^+ \subseteq T_k$. Let $(x, x') \in R_I^+$. There must exist a path over R-edges from x to x'. Let l be the length of the path, i.e., $(x, x') \in R^l$. Note that $R \subseteq T_k$, and thus, $R^l \subseteq T_k^l$. As T_k is transitive, $T_k^l \subseteq T_k$. □

This suggests a trivial algorithm that attempts to construct d.wf. relations T_i for incrementally deep unwindings of P until it finally finds a transitive T_k, which proves termination of P. However, this trivial algorithm need not terminate, even for simple inputs. This is due to the fact that any T_i does not necessarily have to be different from a previous T_j, where $j < i$. In this case the algorithm will never find a compositional transition invariant.

What follows is a variation of this trivial algorithm that does not suffer from these limitations and takes advantage of the fact that most terminating loops encountered in practice have transition invariants with few disjuncts. This requires exclusion of computations of the program that have been proven terminating in a previous iteration. The following lemma provides a basis for this operation:

Lemma 5. *Let $P = \langle S, I, R\rangle$ and $k \geq 1$. Let $T_1, \ldots, T_k$ be a sequence of d.wf. relations such that each is a superset of the respective $\bigcup_{j=1}^{i} R^j$ restricted to reachable transitions that are not contained in any previous T_j, i.e.,*

$$\left(\bigcup_{j=1}^{i} R^j \setminus \bigcup_{j=1}^{i-1} T_j\right) \cap (R^*(I) \times R^*(I)) \subseteq T_i \ .$$

If $Q := \bigcup_{i=1}^{k} T_i$ is transitive, then Q is a compositional transition invariant for the program P.

Proof. We have $\bigcup_{i=1}^{k} R^i \subseteq \bigcup_{i=1}^{k} T_i = Q$ and in particular $R \subseteq Q$. Therefore $R^+ \subseteq Q^+$. Since Q is transitive it follows that $R^+ \subseteq Q$. It is d.wf. as it is a finite union of d.wf. relations. □

As an optimization, intermediate transition invariants may safely be omitted while searching for a compositional T_k:

Lemma 6 (Optimization)**.** *Let $T_0, \ldots, T_k$ be the sequence of d.wf. relations for application of Lemma 5. The claim of the lemma holds even if some of the $T_1, \ldots, T_{k-1}$ are empty.*

Proof. The goal is to show that Q is a transition invariant for P. Let $(x, x') \in R^+ \cap (R^*(I) \times R^*(I))$. As in the proof of Lemma 4, $(x, x') \in R^l$ for some l. The claim holds trivially for $l \leq k$ as $\bigcup_{i=1}^{k} R^i \subseteq Q$. For $l > k$, note that $(x, x') \in (R^{jk} \circ R^{l-jk})$ and $0 \leq l - jk < k$ for some $j \geq 1$. Note that $R^{jk} \subseteq Q^j$ and $R^{l-jk} \subseteq Q$. Thus, $(x, x') \in (Q^j \circ Q) = Q^{j+1}$. As Q is transitive, $Q^{j+1} \subseteq Q$, and thus $(x, x') \in Q$. □

As an example, in the experimental evaluation of this algorithm in Chapter 7, only those T_i where i is a power of two are used.

Algorithm 2 presents Compositional Termination Analysis (CTA). This algorithm constructs termination arguments such that the prerequisites of Lemma 6 are fulfilled whenever it returns ‘Terminating’ at line 10. It makes use of an external ranking procedure called $rank$, generating d.wf. ranking relations for a given set of transitions, or alternatively a set $C \in S$ of states such that $R^*(C)$ contains infinite computations, i.e., C is a set of initial states that allow non-terminating behavior.

```
input  : $P = \langle S, I, R \rangle$
output: ‘Terminating’ / ‘Non-Terminating’
begin
    $T := \emptyset$;
    $X := S$;
    $i := 1$;
    while true do
        $\langle T_i, C \rangle$ := $rank$ $((\bigcup_{j=1}^{i} R^j \setminus T) \cap (X \times X))$;
        if $C \cap R^*(I) \neq \emptyset$ then
            return ‘Non-Terminating’;
        else if $C = \emptyset$ and $T \cup T_i$ is transitive then
            return ‘Terminating’;
        else
            $X$ := $X \setminus C$;
            $T$ := $T \cup T_i$;
            $i$ := $i + j$, where $j > 0$;
        end
    end
end
```

Algorithm 2: Compositional Termination Analysis

The procedure $rank$ is *sound* if it always returns *either:*

- a d.wf. transition invariant T_i for its input and an empty set C, or
- an empty T_i and a non-empty set of states $C \subseteq S$.

It is considered complete if it terminates on every input.

Algorithm 2 maintains a set $X \subseteq S$ that is an over-approximation of the set of reachable states, i.e., $R^*(I) \subseteq X$. It starts with $X = S$ and at $i = 1$. While iterating over i, it generates d.wf. ranking relations T_i for the transitions in $\bigcup_{j=1}^{i} R^j \setminus T$. As such relations continue to be found, they are added to T.

Once the algorithm establishes that T is compositional, the algorithm stops, as P terminates according to Lemma 6. When ranking fails, i.e., when T_i is empty for some i, the algorithm checks whether there is a reachable state in C. In this case $R^*(C)$ contains a counterexample to termination and the algorithm consequently reports P as non-terminating. Otherwise, it removes C from X, which represents a refinement of the current over-approximation of the set of reachable states.

Remark. The check in line 9 of the algorithm corresponds to checking whether $T \circ T \subseteq T$ for some relation $T : S \times S$. This corresponds to checking validity of

$$\forall x, y \in S.\ (x, y) \in T \circ T \Rightarrow (x, y) \in T\ ,$$

which, in the case of symbolically-represented relations, can be established using one call to a suitable decision procedure (usually a SAT or SMT solver).

3.3 Soundness & Completeness

Compositional Termination Analysis crucially depends on an external ranking procedure $rank$. Soundness of the algorithm is affected by the choice of ranking procedure. No checking is carried out that transition invariants T_i returned by $rank$ actually contain the input given to $rank$, and there is no check that disallows non-empty sets C which contain infinite counterexamples. The soundness of the algorithm therefore depends on the soundness of the ranking procedure:

Theorem 4. *Assuming the sub-procedure $rank$ is sound, Algorithm 2 is sound.*

Proof. When the algorithm returns 'terminating' (line 10), the sequence of relations T_i constructed so far is suitable for application of Lemma 6, which proves termination of P (assuming soundness of $rank$). It is easy to see that the set $R^*(I)$ in Lemma 6 can be over-approximated to X, i.e., it applies to

P if it applies to the temporary program $\langle S, X, R \rangle$. If the algorithm returns 'non-terminating' at line 8, it has found a set of reachable states from which infinite computations exist (again assuming soundness of $rank$), i.e., there exists a concrete counterexample to termination. $\square$

Lines 12–14 of Algorithm 2 ensure progress between iterations by excluding unreachable states (C) from the approximation X and adding the most recently found T_i to T. However, for non-terminating input programs, the algorithm may not terminate for two reasons: a) $rank$ is not required to terminate, and b) there may be an infinite number of iterations. This is not the case for finite S, since sound and complete ranking procedures exist (e.g., [33, 81] and the following chapter of this dissertation). Progress towards the goal can thus be ensured:

Corollary 3. *If the sub-procedure $rank$ is sound and complete for finite-state programs, then Algorithm 2 is sound and complete for non-terminating finite-state programs.*

Proof. First, assume a non-terminating input program $P = \langle S, I, R \rangle$. As S is finite there must exist a looping counterexample with a finite stem, i.e., a computation $s_0, s_1, \ldots, s_n$ where $s_n = s_i$ for some $i < n$. In each iteration, either T increases or X decreases, as $C \cap X = \emptyset$ (assuming soundness and completeness of $rank$). Thus, the algorithm will eventually consider an unwinding long enough to contain the stem, at which point $rank$ returns a C with $C \cap R^*(I) \neq \emptyset$ (since it is sound and complete). In both cases, progress is ensured because $rank$ always returns a d.wf. ranking relation or a non-empty set C. In the worst case, the number of iterations is equal to the length of the shortest counterexample to termination. $\square$

Note that the algorithm is not complete for terminating programs even if they are finite-state. This is due to the fact that T is not guranteed to ever become transitive, even if it contains R^+.

3.4 Discussion & Related Work

A consequence of the incompleteness CTA on terminating finite-state programs is that it may in fact perform worse than other termination proving algorithms, i.e., it may not terminate for inputs that other algorithms analyze quickly. In practice however, there is a trade-off between the number of successful termination proofs and the time required to obtain them. Suitable transition invariants are usually small, have few disjuncts and often are compositional. Conversely, cases in which a compositional transition invariant cannot be found are rare and therefore only few termination proofs are lost due to this incompleteness.

The practical advantage of Compositional Termination Analysis is increased performance over other algorithms whenever compositional transition invariants are found (when using the same sub-procedures for ranking and reachability checks). This is due to the bounded number of unwindings analyzed in every iteration of Algorithm 2 (line 6), which enables a considerable increase in performance over running the reachability checker on the original program.

A second advantage of CTA is that it uses an abstraction-refinement approach to track the reachable states of a program. In practice, termination of many loops does not depend on the entry-state of the loop; in fact it may be easier to establish a compositional transition invariant if the rest of the program is not taken into account. Abstraction-refinement, however, is not a new idea. Its advantages have been exploited in many contexts (important examples in this context are reachability checking algorithms based on the counter-example guided abstraction refinement paradigm [24]). Furthermore, this idea has been exploited in the Terminator algorithm in the form of *Weak* Binary Reachability Analysis [35]: In this variation of the algorithm, each loop in a program is first analyzed in isolation (starting from a non-deterministic entry state) and only if termination cannot be proven for one or more loops is the whole program analyzed.

Example 2. *As a demonstration of the runtime-advantage of CTA over the*

Terminator algorithm, consider the following simple program, where $*$ represents non-deterministic choice.

```
int i = *;
while i<255 do
    if * then
        i:=i+1;
    else
        i:=i+2;
    end
end
```

The state space in this example is $S = \mathbb{N}_0$, and i is the only variable. A suitable wf. transition invariant is $\{(i, i') \in S^2 \mid i < i' \wedge i' \leq 256\}$, which is easily generated within a negligible amount of time. The Terminator algorithm subsequently needs to verify the absence of further counterexamples, which requires 14 refinement iterations when the SATABS *predicate abstraction engine [27] is used for the Binary Reachability check. Compositional Termination Analysis returns immediately after synthesizing the ranking relation, because it is transitive.*

Termination analysis has its roots in the work of Turing [98, 99]. Since then, substantial progress has been made in various areas of computer science: logic programming (e.g., [28, 71]), term rewriting-based analysis (e.g., [47, 96]), and functional programming (e.g., [69]).

The algorithm presented here (CTA) focusses on the iterative construction of a termination argument for a full program. However, it does make use of an external sub-procedure for ranking individual paths. Suitable methods have been developed independent of this dissertation (e.g., [17, 30, 31, 81]). Chapter 4 presents a specialized method for finite-state programs.

CTA is largely based on d.wf. transition invariants and the Terminator algorithm [34,35]. The basis for reasoning about transition invariants, including the result that d.wf. transition invariants can be used to show termination [82]. The Terminator algorithm [35] has its theoretical roots in an abstraction-refinement approach to termination proving presented by the same authors [34].

Compositionality of transition invariants is defined by Podelski and Rybalchenko [82]. They remark that a ‘perhaps curious consequence’ of compositionality of a d.wf. transition invariant is that it is also well-founded. CTA depends crucially on this remark, as well-foundedness of a disjunctively well-founded transition invariant is used as a criterion that terminates the analysis.

Cook et al. [32] present a method that under-approximates the weakest precondition of paths to find a condition for termination. This result may be exploited in the context of CTA as well, as it allows for a generalization of path preconditions, e.g., when $rank$ finds a non-empty but unreachable counterexample set C, this set may be enlarged such that a better refinement of the over-approximation X in Algorithm 2 is achieved.

Berdine et al. present an algorithm for proving termination that is based on abstract interpretation [13]. Using an invariance analysis they construct a variance analysis. They use the fact that the transitive closure of a well-founded relation is also well-founded to show that the fixed-point obtained by their analysis is correct. Their result may be used to improve the overall performance of CTA, as it can be modified to generate d.wf. transition invariants via abstraction.

Biere, Artho, and Schuppan propose an encoding of liveness properties into a (binary) assertion [15]. This approach allows proving termination of programs without a ranking procedure. It has been reported to prove termination of programs that require non-linear ranking functions. However, experiments conducted for this dissertation indicate that the reachability checks required by this method are usually very hard (see Chapter 7 and [33]).

Chapter 4

Bit-Vector Ranking Relation Synthesis

Modern termination provers for imperative programs based on the Terminator algorithm, or Compositional Termination Analysis (Chapter 3), compose termination arguments by repeatedly invoking ranking relation synthesis tools. Such synthesis tools are available for numerous domains, including linear and non-linear systems, and data structures. They construct complex termination arguments that reason about the heap as well as linear arithmetic, nonlinear arithmetic, or both.

Efficient synthesis of ranking relations for machine-level bit-vectors, however, has remained an open problem. Today, the most common approach to creating ranking functions over machine integers is to use tools actually designed for real or rational arithmetic. Because such tools do not faithfully model all properties of machine integers (especially overflows), it is possible that invalid ranking relations can be generated (for both, terminating and non-terminating programs), or that existing ranking relations are not found.

```
unsigned char i;
while (i!=0)
  i = i & (i-1);
```

Figure 4.1: Code fragment from the Windows AGP Driver library (`kernel/agplib/init.c`).

Both phenomena lead to incompleteness of termination provers: verification of programs which do terminate might fail.

This chapter considers the termination problem as well as the synthesis of ranking functions for programs written in languages like ANSI-C, C++, or Java. The crucial feature of these languages is that they provide bit-vector arithmetic over fixed-width words (16, 32, 64, or more bit), and usually support both unsigned and signed datatypes (usually represented using the 2's complement).

Two new algorithms for ranking relation synthesis for bit-vector programs are presented: (i) a complete method based on a reduction to Presburger arithmetic, and (ii) a template-matching approach for predefined classes of ranking functions, including an extremely efficient (but incomplete) method based on a transformation to the SAT problem.

To understand the complexity of the problem, consider the following examples. The programs in these examples are extracted from Windows device drivers and illustrate the difficulty of termination checking for low-level code and the intricacies handling overflow arithmetic.

Example 3. *The program in Figure 4.1 iterates for as many times as there are bits set (to true) in the variable* i*. Termination of the loop can be proven by finding a (decreasing)* ranking function*. To find a suitable function for this example, it is necessary to take the semantics of the bit-wise AND operator & into account, which is not easily possible in methods based on real or rational arithmetic. Here, a possible ranking function is the trivial function* $r(\mathtt{i}) = \mathtt{i}$*,*

```
unsigned long ulByteCount;
for (int nLoop = ulByteCount;
     nLoop; nLoop -= 4) { [...] }
```

Figure 4.2: Code fragment from a Windows audio device driver (`audio/gfxswap.xp/filter.cpp`).

```
unsigned char Index;
unsigned int Head, i;

assume(Index != ((Head - 1) & 31));
i = Head;
while (i!=Index)
  i = (i+1) & 31;
```

Figure 4.3: Code fragment from the Windows AC97 driver (`audio/ac97/wavepcistream2.cpp`).

because the result of i & (i−1) is always in the range $[0, i-1]$ (except when $i = 0$, but the loop guard prevents the loop from being entered in this case). Therefore, the value of $r(\mathtt{i})$ decreases with every iteration, but it can not decrease indefinitely as it is bounded from below (at 0, since i is unsigned).

Example 4. *The program in Figure 4.2 is potentially non-terminating, because the variable nLoop might be initialized with a value that is not a multiple of 4, so that the loop condition is never falsified. For a correct analysis, it is necessary to consider that integer underflows do not change the remainder modulo 4. Ignoring overflows, but given the information that the variable nLoop is in the range $[-2^{31}, 2^{31}-1]$ and is decremented in every iteration, a ranking function synthesis tool might incorrectly produce the ranking function $(\mathtt{nLoop}) = \mathtt{nLoop}$.*

Example 5. *The program in Figure 4.3 contains another example of potentially non-terminating bit-vector code. This code does not terminate when* $\texttt{Index} > 31$*. If this invariant is established by the rest of the program leading to the loop, the program terminates and* $r(\texttt{i}) = -\texttt{i}$ *is a suitable ranking function to prove termination.*

The rest of this chapter is organized as follows: Section 4.1 defines bit-vector programs. Section 4.2 presents a method for ranking relation synthesis based on a reduction to Presburger arithmetic. Section 4.3 presents an approach based on template-matching for predefined classes of ranking functions.

The results presented in this chapter were first published in [33].

4.1 Termination of Bit-Vector Programs

While the methods presented here are intended for analysis of programs written in languages like ANSI-C, C++ or Java, the presentation benefits greatly from an abstraction from the concrete syntax and datatypes used in those languages. This section introduces a minimal language for bit-vector programs. Naturally, real-world programs can be reduced to this language. In practice, this is done by the termination analyzer, which may possibly also deal with abstractions and other transformations.

Bit-vector programs consist of only a single loop, possibly preceded by a sequence of statements called the *stem*). Note that this is not a restriction for two reasons: (i) Any program may be converted to a single-loop program (see, e.g., [1]), and (ii) ranking relation synthesis is only required to handle single-loop programs when used in the context of Terminator-like algorithms, because the counterexamples extracted always represent a single-loop program with a stem.

The bit-vector program syntax used here permits guards ($\textsf{assume}\ (t)$), sequential composition ($\beta;\gamma$), choice ($\beta \,\square\, \gamma$), and assignments ($x := t$). Programs operate on global variables $x \in \mathcal{X}$, each of which ranges over the set $\mathbb{B}^{\alpha(x)}$ of bit-vectors of (fixed) width $\alpha(x) > 0$. The syntactic categories of programs, statements, and expressions are defined by the following grammar:

$$\begin{aligned}
\langle \mathit{Prog} \rangle &::= \langle \mathit{Stmt} \rangle \ \mathsf{repeat}\ \{\ \langle \mathit{Stmt} \rangle\ \} \\
\langle \mathit{Stmt} \rangle &::= \mathsf{skip} \,\big|\, \mathsf{assume}\ (\langle \mathit{Expr} \rangle) \,\big|\, \langle \mathit{Stmt} \rangle ; \langle \mathit{Stmt} \rangle \,\big|\, \langle \mathit{Stmt} \rangle \,\square\, \langle \mathit{Stmt} \rangle \,\big|\, x := \langle \mathit{Expr} \rangle \\
\langle \mathit{Expr} \rangle &::= 0_n \,\big|\, 1_n \,\big|\, \cdots \,\big|\, *_n \,\big|\, x \,\big|\, \mathsf{cast}_n(\langle \mathit{Expr} \rangle) \,\big|\, \neg\langle \mathit{Expr} \rangle \,\big|\, \langle \mathit{Expr} \rangle \circ \langle \mathit{Expr} \rangle
\end{aligned}$$

Because the width of variables is fixed and does not change during program execution, it is not necessary to introduce syntax for variable declarations. Expressions $0_n, 1_n, \ldots$ are bit-vector literals of width n, the expression $*_n$ non-deterministically returns an arbitrary bit-vector of width n, and the operator cast_n changes the width of a bit-vector (cutting off the highest-valued bits, or extending with zeros as highest-valued bits). The semantics of the bitwise negation operator $\neg$ and of the binary operators

$$\circ \in \{+, \times, \div, =, <_s, <_u, \ \&\ , \ |\ , \ll, \gg\}$$

follows convention. Adding further operations, e.g., bit-vector concatenation, is straightforward and therefore omitted. When evaluating the arithmetic operators $+$, $\times$, $\div$, $\ll$, and $\gg$, both operands are interpreted as unsigned integers. In the case of the strict ordering relation $<_s$ (resp., $<_u$) the operands are interpreted as signed integers in 2's complement format (resp., as unsigned integers).

The colon operator, as in $t : n$, indicates that the expression t is correctly typed and denotes a bit-vector of width n. In the rest of this chapter, it is always assumed that programs are type-correct. (For the complete set of typing rules see Appendix A.)

The state space of programs $\mathcal{S}$ is defined over a (finite) set $\mathcal{X}$ of bit-vector variables with widths α and consists of all mappings from $\mathcal{X}$ to bit-vectors of the correct width: $\mathcal{S} = \{f \in \mathcal{X} \to \mathbb{B}^+ \mid \forall x \in \mathcal{X} \ . \ f(x) \in \mathbb{B}^{\alpha(x)}\}$. The transition relation defined by a statement β is denoted by $R_\beta \subseteq \mathcal{S} \times \mathcal{S}$. In particular, the transition relation for sequences is defined as $R_{\beta_1;\beta_2}(s, s') \equiv \exists s'' \ . \ R_{\beta_1}(s, s'') \wedge R_{\beta_2}(s'', s')$. The loop $\mathsf{repeat} \, \{ \, \beta \, \}$ is non-terminating unless β contains guards, e.g., $\mathsf{repeat} \, \{ \, \mathsf{assume} \, (x \leq 10_{\alpha(x)}); \ x := x + 1_{\alpha(x)} \, \}$ is the program that counts until x has reached (at least) 10.

Example. Consider the program given in Figure 4.2. Using unsigned arithmetic (and $-4 \equiv 2^{32} - 4 \mod 2^{32}$), the corresponding bit-vector program is

$$\begin{aligned}
&\mathsf{repeat} \; \{ \\
&\quad \mathsf{assume} \; (nLoop \neq 0) \\
&\quad nLoop := nLoop + (2^{32} - 4) \\
&\}
\end{aligned} \tag{4.1}$$

4.1.1 Complexity

A bit-vector program $\beta; \mathsf{repeat} \, \{ \, \gamma \, \}$ *terminates* iff there is no infinite sequence of states $a_0, a_1, a_2, \ldots \in \mathcal{S}$ with $R_\beta(a_0, a_1)$ and $R_\gamma(a_i, a_{i+1})$ for all $i > 0$. Bit-vector programs do not provide a heap or recursion and therefore belong to the class of *constant memory* programs. This means that the memory consumption is defined upfront and does not depend on program inputs. The termination of such programs is decidable, more precisely, it is PSPACE-complete: polynomial memory is needed in the size of the program and the size of the program's available memory.

Lemma 7. *Deciding termination of bit-vector programs is PSPACE-complete in the program length*[1] *plus $\sum_{x \in \mathcal{X}} \alpha(x)$, i.e., the size of the program's available memory.*

[1]The number of characters in the program text. Here, a unary representation is used for the index n of the operators $0_n, 1_n, \ldots, *_n$, and cast_n is assumed.

Proof. The proof is split into two parts: the termination problem for bit-vector programs is first shown to be in PSPACE and then shown to be PSPACE-hard.

Termination is in PSPACE. To see that the termination problem for bit-vector programs is in PSPACE, consider an encoding of a bit-vector program α into a QBF of polynomial size in the size of α and the program's available memory. Because the validity problem for QBFs is in PSPACE [95], this shows that program termination for bit-vector programs is in PSPACE as well. The construction is based on the classical proof that QBF is PSPACE-complete [95] and uses a technique called "squaring abbreviation."

First, assume that the transition relations R_β, R_γ of a bit-vector program β; `repeat` $\{\,\gamma\,\}$ are encoded as quantifier-free boolean formulae $\phi_\beta(x, x')$ and $\phi_\gamma(x, x')$. Note, that the encoding can be chosen such that the size of $\phi_\beta(x, x')$ and $\phi_\gamma(x, x')$ is polynomial in the size of β, γ. Now, a predicate $reach(a, b, n)$ with the intended semantics "the statement γ can reach the state b from state a in at most 2^n steps" may be constructed. A naive recursive definition of $reach(a, b, n)$ is:

$$\begin{aligned} reach(a,b,0) &\equiv a = b \vee \phi_\gamma(a,b) \\ reach(a,b,n) &\equiv \exists c.reach(a,c,n-1) \wedge reach(c,b,n-1) \end{aligned}$$

Expanding $reach(a, b, n)$ in this way will obviously lead to a formula that is exponential in size, but that only contains existential quantifiers. Instead, universal quantifiers may be used to compress the formula:

$$\begin{aligned} reach(a,b,0) &\equiv a = b \vee \phi_\gamma(a,b) \\ reach(a,b,n) &\equiv \exists c.\forall a', b'. \begin{pmatrix} a' = a \wedge b' = c \vee a' = c \wedge b' = b \\ \rightarrow reach(a', b', n-1) \end{pmatrix} \end{aligned}$$

Because there is no right-hand side with more than one occurrence of $reach$, this leads to a QBF of a size polynomial in n and the size of γ defining $reach(a, b, n)$.

The predicate $reach$ can now be used to encode termination as a QBF: due to the finiteness of the state space, it is sufficient to construct a formula that states the absence of *lassos* in the transition graph. Assuming that the state space has 2^n elements (i.e., n is the sum of the bit-widths of the variables declared in the program), the corresponding formula is:

$$\neg\exists a, b, c, d.\ (\phi_\beta(a, b) \wedge reach(b, c, n) \wedge \phi_\gamma(c, d) \wedge reach(d, c, n))$$

Overall, the size of the formula is polynomial in n and the size of β, γ, and the formula can obviously be generated from β, γ in polynomial time.

Note that this encoding is equivalent to expressing the termination property as a safety property (e.g, according to [15]), and subsequent application of the QBF-based Bounded Model Checking technique introduced in [61].

Termination is PSPACE-hard. The proof is via a polynomial reduction of the validity problem for QBF. Suppose $\phi = Q_1 x_1. \cdots Q_n x_n.\psi$ is a QBF, where $Q_i \in \{\forall, \exists\}$ and ψ is a matrix. There is a program of polynomial size and memory consumption (in the size of ϕ) that terminates if and only if ϕ is valid. To this end, let $x_1, \ldots, x_n$ be program variables of bit-width 1, i.e., $\alpha(x_i) = 1$ for $i \in \{1, \ldots, n\}$. The matrix of psi may be considered an expression in the grammar defined in Table 4.1. This is not a restriction, because the language provides the boolean operators $\&$, $|$, and $\neg$.

To write the program checking validity of ϕ, further variables are required: a variable num with $\alpha(num) = \lceil \log_2(n+2) \rceil$, variables $r_1, \ldots, r_n$ with $\alpha(r_i) = 1$, and variables $num_1, \ldots, num_n$ with $\alpha(num_i) = 2$.

The validity checker has the following form (for sake of brevity, the bit-widths of literals like 5_3 are omitted):

```
$num := 1;\ num_1 := 0;\ \cdots;\ num_n := 0$
      repeat {   assume $(num = 0\ \&\ \neg r_1)$
              $\Box\ loop_1\ \Box\ loop_2\ \Box\ \cdots\ \Box\ loop_n$
              $\Box$ (assume $(num = n + 1);\ r_{n+1} := \phi;\ num := n$) }
```

Each of the blocks $loop_i$ is responsible for enumerating the possible values of x_i and evaluating the quantifier Q_i:

```
assume $(num = i)$;
  (   (assume $(num_i = 0);\ num_i := 1;\ num := i + 1;\ x_i := 0$)
    $\Box$ (assume $(num_i = 1);\ num_i := 2;\ num := i + 1;\ r_i := r_{i+1};\ x_i := 1$)
    $\Box$ (assume $(num_i = 2);\ num_i := 0;\ num := i - 1;\ r_i := r_i \circ r_{i+1}$)  )
```

where $\circ = \ \&$ for $Q_i = \forall$, and $\circ = \ |$ for $Q_i = \exists$.

Observe that the size of each $loop_i$ is logarithmic in n (because numbers in the range $0, \ldots, n + 1$ need to be encoded). The size of all $loop_i$ blocks combined is therefore in $O(n \log n)$, and the size of the whole program is in $O(s \log s)$, where s is the size of the matrix.

The memory consumption of the validity checker is

n	variables x_i
$+ \lceil \log_2(n + 2) \rceil$	variable num
$+ n$	variables r_i
$+ 2n$	variables num_i
$= 4n + \lceil \log_2(n + 2) \rceil \in O(n)$	

Finally, it should be observed that polynomial time algorithms for the transformation of QBFs into prenex form, as well as the generation of the validity checker exist. □

4.2 Ranking Relation Synthesis via ILP

Ranking relations for bit-vector programs may be synthesized by transforming the problem to checking satisfiability of a formula in Presburger arithmetic, which is checked using an intermediate Integer Linear Programming problem. The method presented here is an extension (or adaptation) of Podelski and Rybalchenko's method for synthesis of linear ranking functions over the reals or rationals (see Section 2.2.3 and [81]).

The first step is to generalize Theorem 3 to disjunctions of systems of inequalities over the integers. Subsequently, it is shown that programs defined by formulas in Presburger arithmetic subsume bit-vector programs. Then, an algorithm to synthesize linear ranking functions for programs defined in Presburger arithmetic is presented. Finally, the ranking functions that were found define a ranking relation through the usual $<$ operator.

4.2.1 Linear ranking functions over the integers

In order to faithfully encode bit-vector operations like addition with overflow (describing non-convex transition relations), it is necessary to consider disjunctive transition relations R:

$$R(x, x') \;\equiv\; \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i \;, \tag{4.2}$$

where $l \in \mathbb{N}$, $A_i, A'_i \in \mathbb{Z}^{k\times n}, b_i \in \mathbb{Z}^k$, and $x, x' \in \mathbb{Z}^n$ range over integer vectors. Linear ranking functions for such relations can be constructed by solving an implication like (2.1) for each disjunct of the relation, as shown below. There is one further complication, however: Farkas' lemma, which is the main ingredient for Theorem 3, is in general not complete for inequalities over the integers.

Farkas' lemma is complete for *integral* systems, however: $Ax + A'x' \leq b$ is called integral if the polyhedron $\{\binom{x}{x'} \in \mathbb{Q}^{2n} \mid Ax + A'x' \leq b\}$ coincides with its integral hull (the convex hull of the integer points contained in it). Every system of inequalities can be transformed into an integral system with the same integer solutions, although this might increase the size of the system exponentially [89].

This means that Farkas' lemma does *not* hold if the unknowns range over the integers; implied inequalities in this case can in general not be represented as non-negative linear combinations. Farkas' lemma still holds, however, in the special case of *integral* systems of inequalities. A system $Ax \leq b$ is called integral if the polyhedron $\{x \in \mathbb{Q}^n \mid Ax \leq b\}$ coincides with its integral hull (the convex hull of the integer points contained in it).[2]

Lemma 8 (Integral version of Farkas' lemma)**.** *Suppose $A \in \mathbb{Q}^{n\times k}$ is a matrix, $b \in \mathbb{Q}^n$ a vector such that the system $Ax \leq b$ of inequalities is satisfiable* and integral, *$c \in \mathbb{Q}^k$ is a (row) vector, and $\delta \in \mathbb{Q}$ is a rational. Then*

$$\{x \in \mathbb{Z}^k : Ax \leq b\} \subseteq \{x \in \mathbb{Z}^k : cx \leq \delta\} \tag{4.3}$$

if and only if there is a non-negative (row) vector $\gamma \in \mathbb{Q}^n$ such that $\gamma A = c$ and $\gamma b \leq \delta$.

Proof. The goal is to show that (2.2) if and only if (4.3) in the case of an integral system $Ax \leq b$. The conjecture then follows by Farkas' lemma (Lemma 3).

(2.2) $\Rightarrow$ (4.3): holds because of $\mathbb{Z} \subset \mathbb{Q}$.

(4.3) $\Rightarrow$ (2.2): suppose (4.3) holds. This implies that the convex hull of the set $\{x \in \mathbb{Z}^k : Ax \leq b\}$ is contained in the half-space $\{x \in \mathbb{Q}^k : cx \leq \delta\}$. The convex hull of $\{x \in \mathbb{Z}^k : Ax \leq b\}$ is the same as the integral hull of $\{x \in \mathbb{Q}^k : Ax \leq b\}$, which coincides with $\{x \in \mathbb{Q}^k : Ax \leq b\}$ because $Ax \leq b$ is integral. Therefore (2.2). □

Transforming an arbitrary system $Ax \leq b$ into an integral system with the same integer solutions is achieved as follows: first, an equivalent *total dual*

[2]This deviates from the terminology in [89], where integrality is attributed to polyhedra. For the sake of brevity, the same terminology is applied to integral systems of inequalities here.

integral system $A'x \leq b'$ is derived from $Ax \leq b$, such that $A' \in \mathbb{Z}^{n' \times k}$. A system $A'x \leq b'$ is total dual integral if the duality equation

$$\max\{cx : A'x \leq b'\} = \min\{yb : y \geq 0,\, yA' = c\}$$

has an integral optimum solution y for each integral vector c for which the minimum is finite [89]. $A'x \leq b'$ can then be strengthened to $A'x \leq \lfloor b' \rfloor$ without losing integer solutions. Iterating this procedure eventually leads to an integral system of inequalities.

Linear ranking functions for systems of inequalities over integers may be constructed with the help of the integral version Farkas' lemma (Lemma 8) and the transformation to integral systems as follows:

Lemma 9. *Suppose $R(x, x') \equiv \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i$ is a transition relation in which each disjunct is satisfiable and integral. R has a linear ranking function $m(x) = rx + c$ if and only if there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ for $i \in \{1, \ldots, l\}$ such that:*

$$\lambda_1^i A'_i = 0, \quad \lambda_2^i (A_i + A'_i) = 0, \quad \lambda_2^i b_i < 0, \quad (\lambda_1^i - \lambda_2^i) A_i = 0, \quad \lambda_2^i A'_i = r. \tag{4.4}$$

Proof. Suppose $R(x, x') \equiv \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i$ is a transition relation in which each disjunct is satisfiable and integral. Each direction of the equivalence is shown separately:

$\Rightarrow$: Assume the relation $R(x, x')$ has a linear ranking function $m(x) = rx + c$. Arguing as in the proof [82, Theorem 2], this means that for some $\delta > 0$ and all $i \in \{1, \ldots, l\}$ it follows that

$$\text{for all } x, x' \in \mathbb{Z}^n : A_i x + A'_i x' \leq b_i \text{ implies} \\ rx + c \geq 0 \wedge rx' + c \geq 0 \wedge rx' + \delta \leq rx \tag{4.5}$$

By Lem. 8, this implies that there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ such that for $i \in \{1, \ldots, l\}$:

$$\begin{array}{lll} \lambda_1^i A_i = -r, & \lambda_1^i A_i' = 0, & \lambda_1^i b_i \leq c, \\ \lambda_2^i A_i = -r, & \lambda_2^i A_i' = r, & \lambda_2^i b_i \leq -\delta \end{array}$$

It is now easy to see that (9) is implied by these equations and inequalities.

$\Leftarrow$: Assume (9) holds for non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ for $i \in \{1, \ldots, l\}$. By Theorem 3, for each $i \in \{1, \ldots, l\}$ the disjunct $A_i x + A_i' x' \leq b_i$ has a linear ranking function of the form $m_i(x) = rx + c_i$, which implies that

$$m(x) = rx + \min\{c_i : i \in \{1, \ldots, l\}\}$$

is a ranking function for $R(x, x')$. $\square$

Combined, these results allow the formulation of Algorithm 3 which synthesizes ranking functions for disjunctive integer transition relations.

4.2.2 Ranking functions for Presburger arithmetic

Presburger arithmetic (PA) is the first-order theory of integer arithmetic without multiplication [84]. This section presents a complete procedure to generate linear ranking functions for PA-defined transition relations by reduction to Lemma 9.[3]

Suppose a transition relation $R(x, x')$ is defined by a Presburger formula. Because PA allows quantifier elimination [84], it can be assumed that $R(x, x')$ is a quantifier-free Boolean combination of equations, inequalities, and divisibility constraints $\epsilon \mid (cx + dx' + e)$. Divisibility constraints are introduced during quantifier elimination and state that the value of the term $cx + dx' + e$ (with $c, d \in \mathbb{Z}^n, e \in \mathbb{Z}$) is a multiple of the positive natural number $\epsilon \in \mathbb{N}^+$.

[3]The procedure can also derive ranking functions that contain integer division expressions $\lfloor \frac{t}{\epsilon} \rfloor$ for some $\epsilon \in \mathbb{Z}$, but it is not complete for such functions. Assuming that a polynomial method is used to solve (4.4), the complexity of our procedure is singly exponential.

```
Input: Transition relation $R(x,x') \equiv \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i$
Output: Ranking function $m(x) = rx - c$, or "No linear ranking exists"
foreach integral disjunct $A_i x + A'_i x' \leq b_i$ do
    if $A_i x + A'_i x' \leq b_i$ has no ranking function (Theorem 3) then
        return No linear ranking function exists;

end
while true do
    if there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ s.t. (4.4) holds then
        $c \leftarrow \min\{rx' : \exists x.R(x,x')\}$;
        return $m(x) = rx - c$ is a ranking function;
    if all disjuncts $A_i x + A'_i x' \leq b_i$ are integral then
        return No linear ranking function exists;
    Pick a non-integral disjunct $A_i x + A'_i x' \leq b_i$;
    Strengthen $A_i x + A'_i x' \leq b_i$ to $A_i x + A'_i x' \leq \lfloor b_i \rfloor$;
    if $A_i x + A'_i x' \leq b_i$ is integral and has no ranking (Theorem 3) then
        return No linear ranking function exists;

end
```

Algorithm 3: Ranking function synthesis for disjunctive integer transition relations with iterative strengthening

In order to apply Lemma 9, divisibility constraints must be eliminated from $R(x,x')$ as explained below. This is possible by introducing auxiliary program variables y, y' for each divisibility constraint: The formula $R(x,x')$ is transformed to a new formula $R'(x,y,x',y')$ which does not contain divisibility constraints, such that $\exists y, y'.R'(x,y,x',y') \equiv R(x,x')$. The transformation increases the size of the PA formula only polynomially.

By rewriting to disjunctive normalform, replacing equations $s = t$ with inequalities $s \leq t \wedge t \leq s$, the relation $R'(x, y, x', y')$ can be stated as in (4.2):

$$R'(x, y, x', y') \;\equiv\; \bigvee_{i=1}^{l} A_i \begin{pmatrix} x \\ y \end{pmatrix} + A'_i \begin{pmatrix} x' \\ y' \end{pmatrix} \leq b_i$$

Lemma 9 may then be applied to R' to derive a linear ranking function $m'(x, y)$. To ensure that no auxiliary variables y occur in $m'(x, y)$ (i.e., $m'(x, y) = m(x)$), equations are added to (4.4) that constrain the corresponding entries of r to zero.

Replacing divisibility constraints by disjunctions of equations

The following equivalences are used in the transformation from $R(x, x')$ to $R'(x, y, x', y')$:

$$\epsilon \mid (cx + dx' + e) \;\equiv \epsilon \;\Big|\; \Big(cx - \epsilon\Big\lfloor\frac{cx}{\epsilon}\Big\rfloor + dx' - \epsilon\Big\lfloor\frac{dx'}{\epsilon}\Big\rfloor + e\Big) \tag{4.6}$$

$$\equiv \bigvee_{\substack{i \in \mathbb{Z} \\ 0 \leq i \cdot \epsilon - e < 2\epsilon}} i \cdot \epsilon - e \;=\; cx - \epsilon\Big\lfloor\frac{cx}{\epsilon}\Big\rfloor + dx' - \epsilon\Big\lfloor\frac{dx'}{\epsilon}\Big\rfloor \tag{4.7}$$

$$\equiv \;\exists y_c, y'_d.\; \begin{pmatrix} & 0 \leq cx - \epsilon y_c < \epsilon \wedge 0 \leq dx' - \epsilon y'_d < \epsilon \\ \wedge & (\bigvee_{0 \leq i \cdot \epsilon - e < 2\epsilon} i \cdot \epsilon - e = cx - \epsilon y_c + dx' - \epsilon y'_d) \end{pmatrix} \tag{4.8}$$

Equivalence (4.6) holds because divisibility is not affected by subtracting multiples of ϵ on the right-hand side, while (4.7) expresses that the value of the term $cx - \epsilon\lfloor\frac{cx}{\epsilon}\rfloor + dx' - \epsilon\lfloor\frac{dx'}{\epsilon}\rfloor$ lies in the right-open interval $[0, 2\epsilon)$. Therefore, the divisibility constraints of (4.6) are equivalent to a disjunction of exactly two equations. Finally, the integer division expressions $\lfloor\frac{cx}{\epsilon}\rfloor$ can equivalently be expressed using existential quantifiers in (4.8).

To avoid the introduction of new quantifiers, the quantified variables y_c, y'_d are treated as program variables. Whenever a constraint $\epsilon \mid (cx + dx' + e)$ occurs in $R(x, x')$, new pre-state variables y_c, y_d and post-state variables y'_c, y'_d

are introduced, defined by adding conjuncts to $R(x, x')$:

$$\begin{aligned} R'(x, y_c, y_d, x', y'_c, y'_d) \equiv\ & R(x, x') \land 0 \leq cx - \epsilon y_c < \epsilon \land 0 \leq dx - \epsilon y_d < \epsilon \\ & \land 0 \leq cx' - \epsilon y'_c < \epsilon \land 0 \leq dx' - \epsilon y'_d < \epsilon \end{aligned}$$

In $R'(x, y_c, y_d, x', y'_c, y'_d)$, the constraint $\epsilon \mid (cx + dx' + e)$ can then be replaced with a disjunction $\bigvee_{0 \leq i \cdot \epsilon - e < 2\epsilon} i \cdot \epsilon - e = cx - \epsilon y_c + dx' - \epsilon y'_d$ as in (4.8). Iterating this procedure eventually leads to a transition relation without divisibility constraints, such that $\exists y, y'. R'(x, y, x', y') \equiv R(x, x')$.

Representation of bit-vector operations in PA

Presburger arithmetic is expressive enough to capture the semantics of all bit-vector operations defined in Section 4.1, so that ranking functions for bit-vector programs can be generated using the method from the previous section. For instance, the semantics of a bit-vector addition $s + t$ can be defined in weakest-precondition style as:

$$wp(x := s + t, \phi) \;=\; wp\begin{pmatrix} y_1 := s;\ y_2 := t, \\ \exists x.(0 \leq x < 2^n \land 2^n \mid (x - y_1 - y_2) \land \phi) \end{pmatrix}$$

where $s : n, t : n$ denote bit-vectors of length n, and y_1, y_2 are fresh variables. The existentially quantified formula assigns to x the remainder of $y_1 + y_2$ modulo 2^n.

A precise translation of non-linear operations like $\times$ and $\&$ is achieved by case analysis over the values of their operands, which in general leads to formulae of exponential size, but is well-behaved in many cases that are practically relevant (e.g., if one of the operands is a literal). Of course, this encoding is only possible because the variables of bit-vector programs range over finite domains of fixed size.

Example 6. *The bit-vector program* (4.1) *corresponding to Figure 4.2 is en-*

coded in PA as follows:

$$\begin{aligned} &nLoop \neq 0 \ \wedge \ 2^{32} \mid (nLoop' - nLoop - 2^{32} + 4) \\ \wedge \ &0 \leq nloop < 2^{32} \ \wedge \ 0 \leq nloop' < 2^{32} \end{aligned}$$

From the side conditions, it can be seen that $nLoop' - nLoop - 2^{32} + 4$ *has the range* $[5 - 2^{33}, 3]$*, so that the divisibility constraint can directly be split into two equations (auxiliary variables as in* (4.8) *are unnecessary in this particular example). With further simplifications, the transition relation is expressed as:*

$$\begin{aligned} &\big(nLoop' = nLoop - 4 \ \wedge \ 0 \leq nloop' \ \wedge \ nloop < 2^{32}\big) \\ \vee \ &\big(nLoop' = nLoop + 2^{32} - 4 \wedge 0 < nloop \wedge nloop' < 2^{32}\big) \ . \end{aligned}$$

It is now easy to see that each disjunct is satisfiable and integral, which means that Lemma 9 is applicable. Because the conditions (4.4) *are not simultaneously satisfiable for all disjuncts, no linear ranking function exists for this program.*

4.3 Ranking Relation Synthesis via Templates

Ranking functions for bit-vector programs can be identified by templates of a desired class of functions with undetermined coefficients. In order to find the coefficients, two methods are presented here: (i) an encoding into QBF to check all suitable values, and (ii) an encoding into SAT to check likely values.

4.3.1 Arbitrary Ranking Functions

The state space for bit-vector programs is always finite. It is therefore possible to (explicitly) construct the set of all possible ranking functions for bit-vector programs of some fixed set of states. In practice, this may be exploited to search for ranking functions.

Let $R(x, x')$ be the transition relation of some bit-vector program. Both, x and x' constitute bit-vectors of some fixed size n. Every ranking relation for this program is therefore a relation $r : \mathbb{B}^n \times \mathbb{B}^n$ and there are only 2^{2^n} such relations. This set of relations may be represented using the *Algebraic Normal Form (ANF)* of functions with n Boolean variables:

$$\begin{aligned} ANF(c, x) \quad = \quad & c_0 + \\ & c_1 \cdot x_1 + c_2 \cdots x_2 + \cdots + c_n \cdot x_n \\ & c_{1,2} \cdot x_1 \cdot x_2 + \cdots + c_{n-1,n} \cdot x_{n-1} \cdot x_n + \\ & \cdots + \\ & c_{1,2,\ldots,n} \cdot x_1 \cdot x_2 \cdot \cdots \cdot x_n \ , \end{aligned}$$

where each x_i represents a single bit extract from the vector x and all $c = \{_0, c_1, \ldots, c_{1,2,\ldots,n}\}$ are Boolean coefficients. Every Boolean function may be obtained from the ANF by fixing corresponding coefficients. Therefore, a ranking relation may be obtained by finding a solution to

$$\exists c \, \forall x, x' \ . \ R(x, x') \Rightarrow ANF(c, x') < ANF(c, x) \ . \qquad (4.9)$$

It is straightforward to flatten Equation (4.9) into a QBF. Thus, a QBF solver that returns an assignment for the top-level existential variables is able to compute suitable coefficients. However, Equation 4.9 requires a quadratic number of coefficients and it's size is exponential in the size of x. In practice, this is either too large to be constructed or too hard to solve using a QBF solver. Suitable ranking functions for programs that occur in practice are however often relatively simple, which is why the next section considers only a subset of all ranking functions, namely polynomial functions.

4.3.2 Polynomial Ranking Functions

Let $x = (x_1, \ldots, x_{|\mathcal{X}|})$ be a vector of program variables and associate a coefficient c_i with each $x_i \in \mathcal{X}$. The coefficients constitute the vector $c =$

$(c_1, \ldots, c_{|\mathcal{X}|})$. A polynomial template takes the form

$$p(c, x) := \sum_{i=1}^{|\mathcal{X}|} (c_i \times \mathsf{cast}_w(x_i))$$

with the bit-width $w \geq \max_i(\alpha(x_i)) + \lceil \log_2(|\mathcal{X}| + 1) \rceil$ and $\alpha(c_i) = w$, chosen such that no overflows occur during summation. The following theorem provides a bound on w that guarantees that ranking functions can be represented for all programs that have polynomial ranking functions.

Theorem 5. *There exists a polynomial ranking function for the transition relation $R(x, x')$, if*

$$\exists c\, \forall x, x' \,.\, R(x, x') \Rightarrow p(c, x') <_s p(c, x) \,. \tag{4.10}$$

Vice versa, if there exists a polynomial ranking function for $R(x, x')$, then Equation (4.10) must be valid whenever

$$w \geq \max_i(\alpha(x_i)) \cdot (|\mathcal{X}| - 1) + |\mathcal{X}| \cdot \log_2 |\mathcal{X}| + 1 \,.$$

Proof. The proof is via bounding of the number of bits required to represent all polynomial functions. To show that the upper bound is reasonably tight, a lower bound is given first.

Lower bound. Consider terminating programs (for which polynomial ranking functions exist) of the following form:

```
i=1;
while $i \neq 0 \vee j \neq 0 \vee k \neq 0 \ldots$ do
    $\ldots$
    $k := k + (i \div 255) \times (j \div 255);$
    $j := j + i \div 255;$
    $i := i + 1;$
end
```

where the width of all variables is n. Programs built after this scheme require ranking functions that order variables lexicographically. A suitable ranking function is of the form

$$\cdots + c_k \times k + c_j \times j + c_i \times i \;,$$

where $c_i = -1$, $c_j = -2^n$, $c_k = -2^{2n}$, etc. This means that the corresponding bit-widths of the coefficients are $\alpha(c_i) = 2$, $\alpha(c_j) = n + 2$, and $\alpha(c_k) = 2n + 2$. Note that the constant 2 arises from the fact that each coefficient is of the form -2^x for some x, which is not contained in $[0, 2^x)$, and thus an extra bit is required to represent a coefficient of this size and its sign.

The total number of bits required to represent all coefficients is thus

$$\sum_{i=0}^{|\mathcal{X}|-1} i \cdot n + 2 = 2 \cdot |\mathcal{X}| + n \cdot \sum_{i=0}^{|\mathcal{X}|-1} i \;,$$

which may be rewritten to

$$2 \cdot |\mathcal{X}| + n \cdot \frac{|\mathcal{X}| \cdot (|\mathcal{X}| - 1)}{2} \;.$$

I.e., the programs considered require $O(|\mathcal{X}|^2 \cdot n)$ bits.

Upper bound. Consider a transition relation $R(s, s')$ over a vocabulary $\mathcal{X}$ of variables. For sake of simplicity, assume that the bit-width of all $|\mathcal{X}| = m$ variables is n, i.e., $\alpha(x) = n$ for all $x \in \mathcal{X}$ (this means that all variables range over $[0, 2^n)$). States $s \in \mathcal{S}$ are identified with elements of the vector space $\mathbb{Q}^{|\mathcal{X}|}$, for an arbitrary but fixed enumeration $x_1, \ldots, x_m$ of the variables $\mathcal{X}$. The candidates for ranking functions are the elements of the vector space $V = \mathbb{Q}^{|\mathcal{X}|} \to \mathbb{Q}$ of rational linear functions over the program variables $\mathcal{X}$.

Every function $f \in V$ determines an order $\prec_f$ on the state space $\mathcal{S}$:

$$s \prec_f s' \;\equiv\; f(s) < f(s')$$

If $\prec_f$ is a (strict) total order, f is called *perfectly separating*. If two functions f, f' determine the same order, i.e., $\prec_f = \prec_{f'}$, they are *equivalent*. Obviously, as $\mathcal{S}$ is finite, V is partitioned into finitely many equivalence classes in this way. Therefore, the problem may be simplified through the following observations:

- Because functions f, f' in the same equivalence class can prove the termination of exactly the same loops, it is sufficient to consider ranking function templates that represent at least one function from each equivalence class.
- The classes of perfectly separating functions subsume the classes of non-perfect functions, in the sense that every loop that can be proven terminating using the latter can also be proven terminating by perfectly separating functions.

When choosing the bit-width of coefficients in ranking function templates, it is thus sufficient to ensure that the template represents at least one function in each class of perfectly separating functions.

The geometry of equivalence classes. There is a simple geometric interpretation of equivalence classes. A non-perfect function f has the property that $f(s) = f(s')$ for some states $s \neq s'$. Because states are interpreted as vectors, this is equivalent to $f(s - s') = 0$. For any two states $s \neq s' \in \mathcal{S}$, observe that the set $E_{s,s'} = \{f \in V : f(s - s') = 0\}$ is a hyperplane of the vector space V, which altogether means that the set of non-perfect functions is the finite union

$$E = \bigcup_{s \neq s' \in \mathcal{S}} E_{s,s'}$$

of hyperplanes. The inverse $P = V \setminus E$ is then a finite union of open convex sets, each of which forms the interior of a convex (but unbounded) polyhedron.

These polyhedra coincide with the equivalence classes of perfectly separating functions. To see this, note that for each state $s \in \mathcal{S}$ the function $v_s : V \to \mathbb{Q}$, $v_s(f) = f(s)$ is continuous. This implies that if $f, f' \in V$ are perfectly separating functions that are not equivalent, every continuous path from

f to f' in V has to cross E. Furthermore, the classes belonging to different polyhedra are distinct: for each hyperplane $E_{s,s'}$, it holds that $f(s - s') > 0$ for the functions f on one side of the hyperplane, and $f'(s - s') < 0$ for the functions f' on the other (because $f(s - s')$ is linear in f), which implies $s \prec_f s'$ and $s' \prec_{f'} s$.

Choosing representatives. To distinguish a basis of the vector space V, define states $s_1, \ldots, s_m \in \mathcal{S}$ by:

$$s_i(x_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

The basis $\{b_1, \ldots, b_m\} \subseteq V$ of V is then defined (as the dual basis) by:

$$b_i(s_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

This allows the representation of functions in $f \in V$ as $\alpha_1 b_1 + \cdots + \alpha_m b_m$, which intuitively can be understood as

$$f(x_1, \ldots, x_m) \;=\; \alpha_1 x_1 + \cdots + \alpha_m x_m$$

Now, consider linear combinations with integer coefficients $\alpha_1, \ldots, \alpha_m$. Bounds on the absolute values of the coefficients must be derived, such that elements from each class of perfectly separating functions can be represented. These bounds determine the number of bits needed in polynomial ranking function templates.

First, fix a non-empty class $A \subseteq V$ of perfectly separating functions, and assume that $E' \subseteq E$ is the union of all hyperplanes $E_{s,s'}$ that bound A. Each intersection of $m - 1$ such hyperplanes (provided that no two of them are parallel) is a straight line l that is adjacent to A. It remains to be shown that:

(i) Each line l is generated by a function

$$f_l = \alpha_1 b_1 + \cdots + \alpha_m b_m$$

where $|\alpha_i| \leq 2^{n(m-1)} \cdot (m-1)!$ for $i \in \{1, \ldots, m\}$.

(ii) The set A contains a function

$$f_A = \beta_1 b_1 + \cdots + \beta_m b_m$$

where $|\beta_i| \leq 2^{n(m-1)} \cdot m!$ for $i \in \{1, \ldots, m\}$.

To see that (i) holds, note that each hyperplane $E_{s,s'}$ is described by a linear equation $f(s - s') = 0$ that can be expanded to

$$\big(s(x_1) - s'(x_1)\big)\gamma_1 + \cdots + \big(s(x_m) - s'(x_m)\big)\gamma_m = 0$$

if the representation $f = \gamma_1 b_1 + \cdots + \gamma_m b_m$ is chosen. The coefficients are then of the form $s(x_1) - s'(x_1)$ in these equations and they are in the range $[-2^n + 1, 2^n - 1]$. Finding a vector in the intersection of $m - 1$ hyperplanes requires a solution to the following system of $m - 1$ linear equations:

$$\begin{array}{lllll} v_1^1\gamma_1 & + & \cdots & + \, v_m^1\gamma_m & = 0 \\ & & & & \vdots \\ v_1^{m-1}\gamma_1 & + & \cdots & + \, v_m^{m-1}\gamma_m & = 0 \end{array}$$

By elementary algebra, an integer solution to this system can be found by computing the following determinant (S_m is the group of permutations of $\{1, \ldots, m\}$, and the parity $\operatorname{sgn}(\sigma)$ is $+1$ for even and -1 for odd permutations σ):

$$\begin{vmatrix} v_1^1 & \cdots & v_m^1 \\ \cdots & \cdots & \cdots \\ v_1^{m-1} & \cdots & v_m^{m-1} \\ b_1 & \cdots & b_m \end{vmatrix} = \sum_{i=1}^{m} \sum_{\substack{\sigma \in S_m \\ \sigma(m) = i}} \operatorname{sgn}(\sigma) \Big(\prod_{j=1}^{m-1} v_{\sigma(j)}^j \Big) b_i$$

Because of $|v_i^j| < 2^n$ and $|S_m| = m!$, the absolute value of the coefficient of each basis vector b_i on the right-hand side is bounded by $2^{n(m-1)} \cdot (m-1)!$.

For (ii), assume that there are m linearly independent lines $l_1, \ldots, l_m$ (as in (i)) that are adjacent to A. In this case, because A is convex, there is a sum

$$f_A = c_1 f_{l_1} + \cdots + c_m f_{l_m} \in A \; ,$$

with $c_i \in \{-1, +1\}$ for each $i \in \{1, \ldots, m\}$. The bounds on the absolute values of coefficients follow from (i). A similar argument can be used in the case that no m linearly independent lines exist.

From (i), we can derive the number of bits needed for Theorem 5:

$$\log_2(2^{n(m-1)} \cdot m!) \;=\; n(m-1) + \log_2 m! \;\leq\; n(m-1) + m \log_2 m$$

Because both positive and negative coefficients have to be represented, we need a further bit for the sign, which yields the bound $n(m-1) + m \log_2 m + 1$ given in Theorem 5. □

Despite much progress, the capacity of QBF solvers has not yet reached a level where Equations 4.9 or 4.10 can be solved quickly. Therefore, a third, simplistic, option is considered here: enumeration of likely coefficients. First, restrict all coefficients to $\alpha(c_i) = 2$ and fix a concrete assignment $\gamma(c) \in \{0, 1, 3\}$ to the coefficients (corresponding to $\{-1, 0, 1\}$ in 2's complement). Negating and applying γ transforms Equation 4.10 into

$$\neg\exists x, x' \,.\, R(x, x') \wedge \neg(p(\gamma(c), x') <_s p(\gamma(c), x)) \;, \tag{4.11}$$

which is a bit-vector (or SMT QF_BV) formula that may be flattened to a purely propositional formula in the straightforward way. The formula is satisfiable iff p is *not* a genuine ranking function. Thus, we enumerate all possible γ until we find one for which Equation 4.11 is unsatisfiable, which means that $p(\gamma(c), x)$ must be a genuine ranking function for $R(x, x')$. Even though there are $3^{|\mathcal{X}|}$ possible combinations of coefficient values to test and only a very small subset of all polynomial ranking functions can be represented, this method performs surprisingly well in practice (see Chapter 7).

Example 7. *Consider the program given in Fig. 4.1. The only variable in the program is i, and it is 8 bits wide. The symbolic polynomial for this program is $p(c, i) = c \times \mathsf{cast}_9(i)$ with $\alpha(c) = 9$. For the only path through the loop in this example, the transition relation $R(i, i')$ is $i \neq 0 \wedge i' = i \;\&\; (i - 1)$. Solving the* *resulting formula*

$$\exists c \forall i, i' \ . \ R(i,i') \Rightarrow p(c,i') <_s p(c,i)$$

with a QBF-Solver does not return a result within an hour. Alternatively, the formula may be rewritten according to Equation 4.11 to obtain

$$\neg \exists i, i' \ . \ R(i,i') \land \neg(p(c,i') <_s p(c,i)) \ ,$$

which is solved using a SAT-solver for all choices of $c \in \{0,1,3\}$ *in negligible runtime. The formula is unsatisfiable for* $c = 1$*, which means that* $\text{cast}_9(1_9 \times i)$ *is a suitable ranking function. (In this particular example, it is possible to omit the cast, i.e., a suitable ranking function is simply* $r(i) = i$*.)*

4.4 Related work

Numerous efficient methods are now available for finding different classes of ranking functions (e.g., [4, 18, 42, 82]). Some tools are complete for the class of ranking functions for which they are designed (e.g., [82]), others employ a set of heuristics (e.g., [4]). None of these methods supports the generation of ranking functions for machine-level integer programs.

Bradley et al. [18] give a complete search-based algorithm to generate linear ranking functions together with supporting invariants for programs defined in Presburger arithmetic. This is also a key component of the first method presented here (Section 4.2.2), the method proposed here however, uses constraint-based means to synthesize linear ranking functions for programs using Presburger arithmetic. It is worth noting that the methods presented here are decision procedures for the existence of linear (polynomial) ranking functions, while others procedures (e.g., [18]) are sound and complete, but might not terminate when applied to programs for which no linear ranking functions exist.

Ranking function synthesis is not strictly required if the program is a finite-state system. In particular, Biere, Artho and Schuppan describe a reduction of liveness properties to safety by means of a monitor construction [15]. The resulting safety checks require a comparison of the entire state vector whereas, in practice, the safety checks when using ranking functions refer only to few variables. Experimental results indicate that the safety checks for ranking functions are in most cases easier (see [33] and Chapter 7).

Another approach for proving termination of large finite-state systems was proposed by Ball et al. [6]. Their method creates finite abstractions of a system, preserving all properties required to prove termination. Techniques to find suitable abstractions for bit-vector programs are yet to be developed, however. Furthermore, since this technique does not generate ranking relations, it is not clear whether they can be integrated into systems whose aim is to prove termination of programs that mix machine integers with data-structures, recursion, and/or numerical libraries with arbitrary precision, for which mostly ranking relation-based methods exist.

Chapter 5

Certificates for QBF

The validity problem for the logic of quantified Boolean formulas (QBF) is the canonical PSPACE-complete problem. A vast number of problems can be succinctly formulated in QBF. For example, every finite two-player game can be modeled in QBF [45,94]. A multitude of AI planning [49,77,85], and modal logic problems [68] can be formulated in QBF, as well as unbounded and bounded model checking for finite-state systems [16,40,61], and other formal verification problems [10, 49]. In the context of this dissertation, QBF is especially important as a means to synthesize ranking functions (and relations) for bit-vector programs as presented in the previous chapter.

There exist many different flavors of QBF solvers, based on different techniques such as DPLL [22, 48, 70, 79, 87, 104], Q–Resolution [14], BDDs [78], Skolemization [9] or Hyper Binary Resolution [88]. These solvers decide the validity of a QBF, but they do not usually produce any form of proof or certificate for their solutions. As a consequence the solutions that they compute are only of limited utility in applications that require some sort of synthesis, like generating ranking functions or coefficients for ranking relation templates. Furthermore, these state-of-the-art QBF solvers are not reliable enough, i.e., the solutions they compute are often wrong. Results of recent QBF competitions (an annual competition for QBF solvers) demonstrates this fact: the validity of many hard instances had to be “guessed” by means of a majority vote of the contestants and the solvers often disagree [75]. Certificates provide a method to verify these solutions by the means of a (simple) certificate checker, which potentially reduces the number of discrepancies considerably.

Certificates for the decision problem of propositional logic (the SAT problem) are easy to define. For satisfiable instances, solvers provide a satisfying assignment, and in the case of an unsatisfiable instance, a resolution proof is computed. In both cases, the output can be verified easily by means of efficient (polynomial time) and easy-to-inspect proof checkers. The satisfying assignment, or the resolution proof, correspond to a *certificate* of the correctness of the result. In the case of SAT, these certificates serve many additional purposes: for example, satisfying assignments are often used as counterexamples. Resolution proofs are often used as an input for other algorithms. As an example, the computation of Craig interpolants relies on a resolution proof.

A certificate in the context of QBF refers to a *proof* of validity or invalidity of a formula. As in the case of unquantified formulas (SAT), certificates can serve dual purposes: first, they establish trust in the correctness of the result. The second motivation is to use certificates as input to other algorithms. When formulating a two–player game as a QBF, the main goal is of course to determine whether a winning strategy exists. If it does, it is often also interesting to know what the winning strategy is. Besides knowing that the game *can* be won, we want to know *how* to win it. The same holds for invalid formulas: not only do we want to know that a formula is invalid, we also want to find out *why* there is no solution and we wish to convince ourselves of that fact, preferably, in a concise way.

As shown by Tseitin [97], allowing definitions of fresh variables in (unquantified) Boolean formulas can exponentially shorten resolution proofs. The following section presents an extension of Tseitin’s result to the quantified setting. This new *quantified extension rule* enables the definition of a certificate format which allows extension steps alongside resolution steps in proofs (Section 5.2) and is a convenient way to provide models for valid formulas (Section 5.3). Appendix B provides a precise proposal for a certificate format, which allows variable definitions of predefined structure to simplify the extension of QBF solvers with certificate generation code.

The results presented in this chapter were first published in [62].

5.1 The Quantified Extension Rule

The resolution calculus for QBF (see Section 2.1.1) is able to simulate the resolution-based, sound, and complete refutation calculus of Kleine Büning et al. [64]. This also allows tracing refutation proofs of search-based QBF solvers (DPLL [22]) efficiently, even with further optimizations such as trivial falsity [22], SAT and QBF based learning [49,87,104], as well as hyper binary resolution [88].

It is likely that in order to trace other algorithms – for instance, algorithms that are structural or BDD-based – a much stronger proof system is necessary. One of the main contributions in this chapter is to provide this stronger proof system by adding the following quantified extension rule (which is an extension of Tseitin's corresponding rule for the unquantified case [97]).

Definition 27 (Quantified Extension Rule). *Let y be a fresh and existentially quantified variable, which does not occur in the QBF ϕ, and let ψ be a formula that may contain only y and variables in ϕ, where $x \leq y$ for all variables x in ψ. Furthermore, let there be a satisfying assignment for ψ for every assignment to the variables in ϕ. Then the* Quantified Extension Rule *extends ϕ by ψ, i.e., it adds g conjunctively to ϕ. The new variable y is then a* defined variable *and ψ is a* definition *for y with respect to ϕ.*

Note that ψ does not need to enforce a functional dependency of y on other variables in ψ. As a relation, ψ must be total, e.g., it cannot define a partial function, but it may be non-deterministic, e.g., the value of y does not need to be unique. This is a slight generalization of the classical extension rule for propositional logic [97].

In adapting the extension rule to the quantified setting, the crucial question is where to quantify new variables, e.g., how defined variables y are ordered with respect to variables that already occur in the formula ϕ. It seems intuitive that new variables can only be existential, i.e., $\Omega(y) = \exists$. It is also clear that they cannot be outside of the scope of the innermost variable on which they depend. In the following sections it is required that defined variables are quantified as far out as possible, i.e., defined variables are required to be in the scope of the innermost variable they depend on, but not in the scope of any larger variables. More formally, let X be the set of variables in ϕ, let Y be the set of variables in ψ and let $m = max(X \cap Y)$. Then $\forall x \in X \;.\; x \leq m \Rightarrow x < y \wedge x > m \Rightarrow y < x$. Note that this is especially important in the context of models for QBFs.

To enforce that proofs are polynomially checkable, it is possible to fall back to the original idea of Tseitin [97] and restrict the set of functions that can be defined and the way they are encoded in to clauses. Useful functions are the constants, equality, negation, if-then-else and conjunction. However, conjunction is sufficient:

Definition 28 (Restricted Quantified Extension Rule)**.** *Let l, r be two literals over variables in the formula ϕ, and let y be a fresh variable. Let ψ be the conjunction of the following clauses: $(\overline{y} \vee l)$, $(\overline{y} \vee r)$, and $(y \vee \overline{l} \vee \overline{r})$. It is required that $\Omega(y) = \exists$ and that $x \leq y$ for all variables x in ψ. Then formula ϕ can be* extended *by adding ψ while maintaining validity.*

The soundness of the restricted rule follows from the soundness of the generic rule, which is proved next. It turns out that this rule is enough to produce proofs from refutations of BDD-based QBF solvers (like EBDDRES) in linear time.

For the proof of the following theorem a substitution operator with distributivity (resp. cofactoring) over Boolean operators is required. Substitution is defined as $\phi\{x \leftarrow c\}$, which denotes the formula ϕ where all occurrences of x are replaced by c and it is required that the following holds for the substitution operator:

Lemma 10. $(\phi \wedge \psi)\{x \leftarrow c\} \equiv \phi\{x \leftarrow c\} \wedge \psi\{x \leftarrow c\}$

The following theorem shows that adding definitions is sound and does not change the semantics of a formula.

Theorem 6 (Soundness of Quantified Extension Rule)**.** *Let ψ be a definition for y with respect to ϕ. Then $[\![\phi]\!] = [\![\phi \wedge \psi]\!]$.*

Proof. The proof is by induction on the number of variables in $\phi \wedge \psi$. Let x be the outermost variable in $\phi \wedge \psi$. First, assume that x is different from y, the variable defined by ψ, and $\Omega(x) = \exists$. The definitions and the lemma imply:

$$\begin{aligned}
[\![\phi \wedge \psi]\!] &= [\![(\phi \wedge \psi)\{x \leftarrow 0\}]\!] \vee [\![(\phi \wedge \psi)\{x \leftarrow 1\}]\!] \\
&= [\![\phi\{x \leftarrow 0\} \wedge \psi\{x \leftarrow 0\}]\!] \vee [\![\phi\{x \leftarrow 1\} \wedge \psi\{x \leftarrow 1\}]\!] \\
&= [\![\phi\{x \leftarrow 0\}]\!] \vee [\![\phi\{x \leftarrow 1\}]\!] \;=\; [\![\phi]\!]
\end{aligned}$$

In the next to last step we have to apply the induction hypothesis twice for the definitions $\psi\{x \leftarrow c\}$ with respect to $\phi\{x \leftarrow c\}$. The second case with $\Omega(x) = \forall$ is identical, except that all the disjunctions are replaced by conjunctions.

For the base case, assume that $x = y$. From the definition of the extension rule, it follows that $\Omega(x) = \exists$, and that x does not occur in ϕ. Since all variables x in ψ have to be smaller or equal to y, ψ is either constant or only contains y as variable. Therefore ψ is either equivalent to $\top$, to the literal $\neg x$, or to the literal x, as otherwise it would be impossible to satisfy ϕ by assigning a value to x. If $\phi \equiv \top$, then $[\![\phi \wedge \psi]\!] = [\![\phi \wedge \top]\!] = [\![\phi]\!]$. Without loss of generality, assume $\psi \equiv x$. Then

$$\begin{aligned}
[\![\phi \wedge \psi]\!] &= [\![(\phi \wedge \psi)\{x \leftarrow 0\}]\!] \vee [\![(\phi \wedge \psi)\{x \leftarrow 1\}]\!] \\
&= [\![\phi\{x \leftarrow 0\} \wedge \psi\{x \leftarrow 0\}]\!] \vee [\![\phi\{x \leftarrow 1\} \wedge \psi\{x \leftarrow 1\}]\!] \\
&= [\![\phi \wedge \psi\{x \leftarrow 0\}]\!] \vee [\![\phi \wedge \psi\{x \leftarrow 1\}]\!] \\
&= [\![\phi \wedge 0]\!] \vee [\![\phi \wedge 1]\!] \;=\; [\![\phi]\!]
\end{aligned}$$

□

5.2 Extended Resolution Proofs

The quantified extension rule allows the definition of a new proof system which contains not only the Q-resolution operation, but also the quantified extension rule. Let C and D be clauses in the matrix of a formula ϕ and let ψ be a definition with respect to ϕ. The proof system is then defined as

$$\frac{C \vee x \qquad D \vee \bar{x}}{forall - reduce(C \vee D)} \text{ Q-Resolution} \qquad \frac{\phi \wedge \psi}{\psi} \text{ Q-Extension}$$

Some operations which are frequently used by QBF solvers allow reasoning across quantifier alternations. Examples are universal expansions (in Quantor [14]) and long-distance resolution (in Quaffle [104]). Universal expansion allows, in principle, a resolution of clauses on outer variables that stem from different copies of the expanded clauses. Similarly, long-distance resolution for learned clauses allows to compactly capture the effect of propagating information from an outer existential scope through a universal quantifier into an inner existential scope and back to the outer existential scope.

Currently, it is not clear whether these operations can be efficiently simulated using the proof system given here. It is possible that an even stronger proof system is needed to achieve this.

5.3 QBF Models

The extended proof system presented in the previous section provides a convenient way to obtain proofs for invalid QBFs. For the case of valid formulas ϕ, the straightforward solution is to give a proof of invalidity of $\neg\phi$. Experiments in which a formula was negated (obtaining a formula in disjunctive normal form) and translating back into a CNF using the Tseitin-transformation have shown that this approach is infeasible in practice. All of the problems that could be solved within 600 seconds by three different solvers (QUANTOR, sKIZZO, SQUOLEM) could either not be solved within the same time when inverted, or took considerably more time to solve. (See Chapter 7 for a more detailed account of this experiment.)

However, the quantified extension rule allows for a convenient definition of models for QBF. In the unquantified case, a certificate for satisfiability is an assignment to the variables which makes the formula evaluate to true. In the case of QBFs, a satisfying assignment for an (existential) variable x depends on the universal variables x is in the scope of. Therefore, an assignment is a set of *functions* instead of a set of values. In the general case, such functions are called *Skolem-functions*.

The notion of "satisfying functions" is exploited here to define QBF models:

Definition 29 (Model). *Let V_i be the set of variables in a formula that have a quantification level less than or equal to i and let E_i and A_i be the sets of existentially resp. universally quantified variables in V_i, i.e., $E_i \cup A_i = V_i$. A model M of the formula is then a set of functions*

$$M := \{f_{v_k} : \mathbb{B}^{k-1} \to \mathbb{B} \mid v_k \in E_n\},$$

where every f_{v_k} depends exactly on the $k-1$ variables from V_{k-1}.[1]

[1] It is also possible to let the f_{v_k} depend only on the universally quantified variables of V_{k-1}. However, the definition given here may result in more compact representations of the functions f_{v_k} in practice.

This definition is essentially the same as the one used by Kleine Büning and Zhao [65]. It is also used in the QBF solver sKIZZO [11] to certify satisfiable resp. valid instances. Using the extension rule, however, it is possible to provide a model as a set of extensions to the formula by defining new variables representing the values of the Skolem functions. A certificate for the formula ϕ thus contains a series of extensions to ϕ, followed by a list of pairs of the form (v, y_v) where y_v is the fresh variable that encodes the Skolem-function for v. As a final step, adding equivalences $v = x_v$ to ϕ for all variables for which a model function is provided makes the formula evaluate to $\top$ if the model is indeed a correct one.

As mentioned before, new variables have to be ordered carefully. If they are simply quantified in the innermost scope, the corresponding function could depend on variables with higher quantification level than the one that it serves as a model for. For example, in a formula with the quantification sequence $\exists x \forall a \exists y$, the newly defined variable y may depend on the value of a, which may turn the defined function into an invalid model. Therefore, extension variables must be quantified as low as possible, i.e., immediately after the variable with the highest quantification level occurring in the extension function. Checking that a model function does not depend on higher quantification levels is then trivial, i.e., fast in practice.

However, checking the *validity* of a model according to Def. 29 is a co-NP complete problem [21, 65]. In practice, it is advisable to check each clause is tautological individually, leveraging incremental SAT solver technology. This can be achieved by keeping the model functions in a SAT solver that supports incremental solving and adding the negation of a clause, i.e., the negations of the clause's literals, as assumptions. The resulting problem must then be unsatisfiable, which means that it is impossible to satisfy the given clause with the model provided. It is possible to provide a refutation proof for each of those sub-problems (or, alternatively, for the whole formula instead of separate clauses). The complexity of checking this "annotated" model for validity is then polynomial in the size of the certificate.

In Skolemization-based solvers constructing a model is easy, since the solvers strategy is in fact to construct a model; it just has to dump the Skolem-functions that it generates (e.g., in the form of BDDs). DPLL- and search-based solvers split on variables using different strategies. Assuming the formula is valid, the solver always encounters either unit clauses, and propagates this information, or splits on a new variable. Whenever a unit clause (l) occurs under some variable assignment α to other variables, it can be read as $\alpha \Rightarrow l$, which corresponds to one entry in the function table of the model function for the variable of l. This information can (immediately) be stored as $x_i = a_1 \wedge ... \wedge a_n$, where x_i is a fresh variable and the a_i are the literals from α. When solving has finished, the final model-function is $f_v = x_1 \vee \cdots \vee x_n$.

In Q-Resolution-based solvers (particularly [87, 88]), one can interpret any resolution between two clauses C and D into a resolvent as the generation of a conflicting clause. For instance, if C contains a literal x and D contains $\neg x$, then $\neg(C \backslash \{x\}) \wedge \neg(D \backslash \{\neg x\})$ may not happen, because x would have to be $\top$ and $\bot$ at the same time to satisfy both clauses. One strategy to record a model is to start with over-specified functions and to refine them whenever resolution is applied. BDD-based solvers implementing the bucket algorithm [78] (and also EBDDRES) store intermediate BDDs for variables to be eliminated. From these, it is possible to dump Skolem-functions using the extension rule. The procedure for EBDDRES is presented in more detail in Chapter 7.

5.4 Discussion & Related Work

Currently only two suggestions for QBF certificates exist: One in the form of an “inference log”, from which a BDD–based model or a refutation for a QBF can be reconstructed [12], and a method for generating unsatisfiable cores from traces of search–based solvers [103]. (In the QBF setting—similar to propositional logic—an unsatisfiable core is a subset of the matrix of an invalid

QBF formula, which is again invalid.)

The first approach does not scale well in practice because of the growth of the BDDs. In fact, for some instances it takes considerably more time to reconstruct the model from the inference log than it took to generate the model in the first place [12]. The inference log that is provided can serve as a refutation trace. However, due to five different inference strategies that the solver (SKIZZO) may choose from, there are many different kinds of instructions in the inference logs. Among them are context switches between inference styles, explicit and symbolic variants of inference rules, such as resolution, substitution, and assignment, but also rollback and commit operations to undo earlier operations, as well as “other control information” [12]. This set of instructions, tailored to keep the overhead of the solver as small as possible, results in the need for a heavyweight proof checker.

The second approach provides only an unsatisfiable core and depends on the particular QBF solver used. In essence, both approaches only “trace”, what the solver is doing and are thus far from a unified format that could be used to certify the computation of QBF solvers based on very different algorithms.

However, the method proposed here is only a first step towards a uniform format. It clearly lacks the ability to capture important features of certain QBF solvers, such as *long distance resolution* [104], and *expansion* [9, 14]. It is entirely possible that the quantified extension rule is not enough to linearly trace the proof process of such solvers.

Chapter 6

Solving Quantified Bit-Vector Formulas

The complexity of integrated circuits continues to grow at an exponential rate and so does the size of the verification and synthesis problems arising from the hardware design process. To tackle these problems, bit-precise decision procedures are a requirement and frequently the crucial ingredient that defines the efficiency of the verification process.

Recent years have seen an increase in the utility of bit-precise reasoning in the area of software verification where low-level languages such as C or C++ are concerned. In both areas, hardware and software design, methods of automated synthesis (e.g., LTL synthesis [80]) become more and more tangible with the advent of powerful and efficient decision procedures for various logics, most notably SAT and SMT solvers. In practice, however, synthesis methods are often incomplete, bound to very specific application domains, or simply inefficient.

In the case of hardware, synthesis usually amounts to constructing a module that implements a specification [60, 80], while for software this can take different shapes: inferring program invariants [53], finding ranking functions for termination analysis [33, 81, 99] (and Chapter 4 of this dissertation), program fragment synthesis [92], or constructing bugfixes following an error-description [93] are all instances of the general synthesis problem.

In this chapter, a new approach to solving quantified bit-vector logic is presented. This logic defines a superset of SAT and QBF formulas and allows for a direct mapping of many hardware and (finite-state) software verification problems and is thus ideally suited as an interface between the verification or synthesis tool and the decision procedure.

In many practically relevant applications, support for uninterpreted functions is not required. In this restricted case, quantified bit-vector formulas are essentially equivalent to QBF. In practice however, QBF solvers face performance problems and they are not usually able to produce models for satisfiable formulas, which is crucial in synthesis applications. The same holds true for many automated theorem provers. SMT solvers on the other hand are efficient and produce models, but usually lack complete support for quantifiers.

The ideas in this chapter combine techniques from automated theorem proving, SMT solving and synthesis algorithms. A set of simplifications and rewriting techniques is proposed, transforming the input formula into a set of equations that an SMT solver is able to solve efficiently. A model finding algorithm is then employed to refine a candidate model iteratively, while function (or circuit) templates are used to reduce the number of iterations required by the algorithm.

The results presented in this chapter were first published in [101].

6.1 Preliminaries

The usual notions and terminology of first order logic and model theory are adopted. The main interest lies on many-sorted languages and bit-vectors of different sizes correspond to different sorts. For each bit-vector sort of size n, the equality $=_n$ is interpreted as the identity relation over bit-vectors of size n. The if-then-else (multiplexer) bit-vector term ite_n is interpreted as usual, i.e., as $ite(true, t, e) = t$ and $ite(false, t, e) = e$. As a notational convention, the subscript is usually omitted. The 0-arity function symbols are called *constant* symbols, and 0-arity predicate symbols are called *propositions*. *Atoms*, *literals*, *clauses*, and *formulas* are defined in the usual way. Terms, literals, clauses and formulas are called *ground* when no variable appears in them. A *sentence* is a formula in which no free variables occur. CNF formulas are denoted as sets of clauses. The symbols a, b and c are used for constants, f and g for function symbols, p and q for predicate symbols, x and y for variables, C for clauses, φ for formulas, and t for terms. The term $x\colon n$ denotes that variable x is a bit-vector of size n. When the bit-vector size is not specified, it is implicitly assumed to be 32. The expression $f\colon n_1, \ldots, n_k \to n_r$ specifies that function symbol f has arity k, it's argument bit-vectors have sizes n_1, ..., n_k, and the result bit-vector has size n_r.

Dependencies are denoted as $\varphi[x_1, \ldots, x_n]$, which means that the formula ϕ may contain variables x_1, ..., x_n. Similarly $t[x_1, \ldots, x_n]$ is defined for terms t. Where there is no confusion, $\varphi[x_1, \ldots, x_n]$ is abbreviated as $\varphi[\overline{x}]$ and $t[x_1, \ldots, x_n]$ as $t[\overline{x}]$. In the rest of this chapter, the difference between functions and predicates is trivial; only functions are therefore discussed.

The standard notion of a structure (interpretation) is used. A structure that satisfies a formula F is called a *model* for F. A theory is a collection of first-order sentences. Interpreted symbols are those symbols whose interpretation is restricted to the models of a certain theory. A symbol is uninterpreted if its interpretation is not restricted by a theory. We use $BitVec$ to denote the bit-vector theory. The usual interpreted symbols for bit-vector theory are used: $+_n$, $*_n$, $concat_{m,n}$, $\leq_n$, 0_n, 1_n, $\ldots$. Where there is no confusion, the subscript specifying the size of the bit-vector is omitted.

A formula is *satisfiable* if and only if it has a model. A formula F is satisfiable modulo the theory $BitVec$ if there is a model for $\{F\} \cup BitVec$.

6.2 Quantified Bit-Vector Formulas

A *Quantified Bit-Vector Formula* (QBVF) is a many sorted first-order logic formula where the sort of every variable is a bit-vector sort. The QBVF-satisfiability problem, is the problem of deciding whether a QBVF is satisfiable (modulo the theory of bit-vectors). This problem is decidable because every universal (existental) quantifier can be expanded into a conjunction (disjunction) of potentially exponential, but finite size. A distinguishing feature of QBVFs is the support for uninterpreted function and predicate symbols. This is useful in many applications, as illustrated by the following example:

Example 8. *Arrays can be easily encoded in QBVF using quantifiers and uninterpreted function symbols. In the following formula, the uninterpreted functions f and f' are used to represent arrays from bit-vectors of size 8 to bit-vectors of the same size, and f' is essentially the array f updated at position $a+1$ with value 0:*

$$f'(a+1) = 0 \ \wedge \ (\forall x : 8.\, x = a + 1 \vee f'(x) = f(x)) \ .$$

Deciding whether a QBF is valid, is a PSPACE-complete problem. Note that any QBF problem can be easily encoded in QBVF by using bit-vectors of size 1. The converse is not true, as QBVF is more expressive than QBF. For instance, uninterpreted function symbols can be used to simulate non-linear quantifier prefixes. The fragment of first-order logic called EPR (effectively propositional logic, or the Bernays-Schoenfinkel class of formulas), comprises formulas of the form $\exists^*\forall^*\varphi$, where φ is a quantifier-free formula with predicates but without function symbols. EPR is a decidable fragment because the Herbrand universe of an EPR formula is always finite. The satisfiability problem for EPR is NEXPTIME-complete, which is the same for QBVF:

Theorem 7. *The satisfiability problem for QBVF is NEXPTIME-complete.*

Proof. The proof consists of showing that there is a polynomial reduction from EPR to QBVF and vice-versa.

QBVF $\Rightarrow$ EPR. Given a QBV formula φ, w.l.o.g. we assume φ is in CNF. The first step is to flatten every clause in φ. The idea is to avoid nested terms by introducing auxiliary variables. Given a clause $\forall \overline{x}.\ C[t]$, where t is a nested term, it is converted into $\forall \overline{x}, y.\ y \neq t \vee C[y]$. Flattening is applied until all literals in a clause are *shallow*. For example, the clause $\forall x_1, x_2.\ f(x_1, g(x_2)) \leq g(x_1)$ is reduced to

$$\begin{aligned}\forall x_1, x_2, y_1, y_2, y_3.\ y_1 \neq g(x_2) \vee y_2 \neq f(x_1, y_1) \vee \\ y_3 \neq g(x_1) \vee y_2 \leq y_3\ .\end{aligned}$$

Next, for each uninterpreted function f where the range is a bit-vector of size n, we create n predicates $p_{f_1}, \ldots, p_{f_n}$. Each bit-vector variable and constant is broken into single bits. A disequality of the form $x \neq f(y)$ is encoded as

$$\begin{aligned}&((x_1 = \top)\ xor\ p_{f_1}(y_1, \ldots, y_n)) \vee \\ &\ldots \\ &((x_m = \top)\ xor\ p_{f_m}(y_1, \ldots, y_n))\ .\end{aligned}$$

Other atoms are encoded in a similar way. Two special constants $\bot$ and $\top$, the axiom $\bot \neq \top$ are added and for each new bit constant c, the clause $c = \bot \vee c = \top$ is added. For example, in the following QBV formula, assume all sorts are bit-vectors of size 2:

$$(\forall x.\ f(f(x)) = 0) \wedge f(a) = 2\ .$$

After flattening, this becomes

$$(\forall x, y.\ y \neq f(x) \vee f(y) = 0) \wedge f(a) = 2\ .$$

Then, after breaking the bit-vectors into single bits:

$$\begin{aligned}
(\forall x_1, x_2, y_1, y_2.\ & ((y_1 = \top)\ xor\ p_{f_1}(x_1, x_2)) \vee \\
& ((y_2 = \top)\ xor\ p_{f_2}(x_1, x_2)) \vee \\
& (\neg p_{f_1}(y_1, y_2) \wedge \neg p_{f_2}(y_1, y_2))) \wedge \\
\neg p_{f_1}(a_1, a_2)\ & \wedge\ p_{f_2}(a_1, a_2) \wedge \\
(a_1 = \top \vee a_1 & = \bot) \wedge \\
(a_2 = \top \vee a_2 & = \bot) \wedge \\
\top \neq \bot\ . &
\end{aligned}$$

EPR $\Rightarrow$ QBVF. Any satisfiable EPR formula has a finite Herbrand model. Moreover, a formula containing n constants has a model with a universe of size at most n. Therefore, in principle, it should be straightforward to reduce an EPR formula to QBVF. In principle, this only requires a bit-vector sort of size $\lceil log_2 n \rceil$. The main problem in this approach is that the EPR formula may contain cardinality constraints such as $\forall x.\ x = a_1 \vee \ldots \vee x = a_m$. For example, this clause is only satisfiable in a model with a universe with size at most m. Now, assume a formula φ with n constants and containing a cardinality constraint limiting the universe size to m. If $m < \lceil log_2 n \rceil$, then the QBVF

$$\forall x : \lceil log_2 n \rceil.\ x = a_1 \vee \ldots \vee x = a_m$$

is equivalent to $false$. This problem can be avoided by using an approach found in several EPR solvers that do not have support for $=$. These solvers use the fact that any EPR formula φ containing $=$ is equisatisfiable to another EPR formula φ' that does not contain $=$. The basic idea is to replace $=$ with a new binary predicate $isEq$, and include the axioms of equality for it:

$$\begin{aligned}
&\forall x. && isEq(x, x) \\
&\forall x, y. && \neg isEq(x, y) \vee isEq(y, x) \\
&\forall x, y, z. && \neg isEq(x, y) \vee \neg isEq(y, z) \vee isEq(x, z) \\
&\forall \overline{x}, \overline{y}. && \neg isEq(x_1, y_1) \vee \ldots \vee \neg isEq(x_n, y_n) \vee \neg p(\overline{x}) \vee p(\overline{y})\ .
\end{aligned}$$

In fact the last axiom is an axiom scheme and one of them is required for each predicate p in the formula φ. □

QBVF can be used to compactly encode many practically relevant verification and synthesis problems. In hardware verification, a fixpoint check consists of deciding whether k unwindings of a circuit are enough to reach all states of the system. To check this, two copies of the k unwindings are used: Let $T[x, x']$ be a formula encoding the transition relation and $I[x]$ a formula encoding the initial states of a circuit. Furthermore, let

$$T^k[x, x'] \equiv T[x, x_0] \wedge \bigwedge_{i=1}^{k-1} T[x_{i-1}, x_i] \wedge T[x_{k-1}, x'] \ .$$

Then a fixpoint check for k unwindings corresponds to the QBV formula

$$\forall x, x' \ . \ I[x] \wedge T^k[x, x'] \rightarrow \exists y, y' \ .I[y] \wedge T^{k-1}[y, y'] \ ,$$

where x, x', y, and y' are (usually large) bit-vectors.

Of renewed interest is the use of symbolic reasoning for code synthesis [92], loop invariants [29, 53] and ranking functions (Chaper 4 and [33]) for finite-state programs. All these applications can be easily encoded in QBVF, as illustrated by the following example:

Example 9. *Consider the following abstract program:*

```
$pre(x)$
**while** $(c(x))$ { $x' := T(x)$ }
$post(x)$
```

In the loop invariant synthesis problem, the goal is to synthesise a predicate I that can be used to show that $post$ holds after execution of the while-loop*. Let, $pre[x]$ be a formula encoding the set of states reachable before the beginning of the loop, $c[x]$ be the encoding of the entry condition, $T[x, x']$ be the transition relation, and $post[x]$ be the encoding of the desired property. Then,* *a suitable loop invariant exists if the following QBV formula is satisfiable:*

$$\begin{aligned}&\forall x.\ pre[x] \rightarrow I(s) \wedge\\&\forall x, x'.\ I(x) \wedge c[x] \wedge T[x, x'] \rightarrow I(x') \wedge\\&\forall x.\ I(x) \wedge \neg c[x] \rightarrow post[x]\ .\end{aligned}$$

An actual invariant can be extracted from any model that satisfies this formula.

Similarly, in the ranking function synthesis problem, the goal is to synthesize a function $rank$ that decreases after each loop iteration and that is bounded from below. This problem can be encoded as the following QBVF satisfiability problem:

$$\begin{aligned}&\forall x.\ rank(x) \geq 0 \wedge\\&\forall x, x'.\ c[x] \wedge T[x, x'] \rightarrow rank(x') < rank(x)\ .\end{aligned}$$

Note that the general case of this encoding requires uninterpreted functions. The call to $rank$ can not be replaced with an existentially quantified variable, as it is impossible to express the correct variable dependencies in a linear quantifier prefix, as is required, e.g., by QBF.

6.2.1 Formula Simplification

Modern first-order theorem provers spend a great deal of their time in simplification or contraction operations. These operations are inferences that remove or modify existing formulas. To solve QBVFs, several simplification and contraction rules found in first-order provers are suggested here and new rules that are particularly useful in common application domains are proposed. These rules help to greatly reduce the size and complexity of typical QBV formulas in practice.

6.2.2 Miniscoping

Miniscoping is a well-known technique for minimizing the scope of quantifiers [54]. In implementations this is often applied after converting the formula to negation normal form (as is the case in the implementation used for the experimental evaluation in Chapter 7). The basic idea is to distribute universal (existential) quantifiers over conjunctions (disjunctions). Formally, the universal case of this simplification rule is

$$(\forall \overline{x}.F[\overline{x}] \wedge G[\overline{x}]) \Longrightarrow (\forall \overline{x}.F[\overline{x}]) \wedge (\forall \overline{x}.G[\overline{x}]) \ .$$

The scope of a quantifier may also be limited if a sub-formula does not contain the quantified variable. That is,

$$(\forall \overline{x}.F[\overline{x}] \vee G) \Longrightarrow (\forall \overline{x}.F[\overline{x}]) \vee G$$

when G does not depend on x.

These transformations are particularly important, because it increases the applicability of rules based on rewriting and macros.

6.2.3 Skolemization

Many first-order theorem provers, eliminate existentially quantified variables using *Skolemization*. This transformation converts the formula $\forall x.\ \exists y.\ \neg p(x) \vee q(x, y)$ to the equisatisfiable formula $\forall x.\ \neg p(x) \vee q(x, f_y(x))$, where f_y is a fresh function symbol.

6.2.4 Conjunction of universally quantified formulas

Once NNF conversion, miniscoping and skolemization have been applied, it is helpful to write a QBV formula as a conjunction of universally quantified formulas:

$$(\forall \overline{x}.\ \varphi_1[\overline{x}]) \wedge \ldots \wedge (\forall \overline{x}.\ \varphi_n[\overline{x}]) \ .$$

This form is very similar to that used in first-order theorem provers. These solvers usually require each $\varphi_i[\overline{x}]$ to be a clause, which is not required here. Some of the $(\forall \overline{x}.\ \varphi_i[\overline{x}])$ may also be ground, that is, $\overline{x}$ is empty.

Note that this conjunctive form is important because it enables many of the other simplifications described below.

6.2.5 Destructive Equality Resolution (DER)

DER allows the fixing of a solution for a negative equality literal by application of the following simple transformation:

$$(\forall x, \overline{y}.\ x \neq t \vee \varphi[x, \overline{y}]) \Longrightarrow (\forall \overline{y}.\ \varphi[t, \overline{y}])\ ,$$

where t does not contain x. For example, using DER, the formula $\forall x, y.\ x \neq f(y) \vee g(x, y) \leq 0$ is simplified to $\forall y.\ g(f(y), y) \leq 0$. DER is essentially an equality substitution rule. This becomes clear when the clause on the left-hand-side is written using an implication, i.e., as

$$\forall x, \overline{y}.\ (x = t) \Rightarrow \varphi[x, \overline{y}]\ .$$

It is straightforward to implement DER; a naive implementation eliminates a single variable at a time. However, in benchmarks with hundreds of variables to be eliminated, this naive implementation may become a bottleneck. The natural solution is to eliminate as many variables simultaneously as possible. The only complication in this approach is that some of the variables being eliminated may *depend* on each other. A variable x *directly depends* on y, when there is a literal of the form $x \neq t[y]$.

In general DER is applicable to formulas of the form

$$\forall x_1, \ldots, x_n, \overline{y}.\ x_1 \neq t_1 \vee \ldots \vee x_n \neq t_n \vee \varphi[x_1, \ldots, x_n, \overline{y}]\ ,$$

where each x_i may depend on variables x_j, $j \neq i$. Let G be a dependency graph where the nodes are the variables x_i, and G contains an edge from x_i to x_j whenever x_j depends on x_i. Topological sorting on G computes an ordering of the nodes in which variables occur only after all of their dependencies. In the case where a cycle is detected during sorting, the corresponding node x_i may simply be removed from G while at the same time

$x_i \neq t_i$ is moved to $\varphi[x_1, \ldots, x_n, \overline{y}]$. Finally, the variable order $x_{k_1}, \ldots, x_{k_m}$ ($m \leq n$) computed is used to apply multiple DER operations simultaneously. Let θ be a *substitution*, i.e., a mapping from variables to terms. Initially, θ is empty. For each variable x_{k_i}, apply θ to t_{k_i} producing t'_{k_i}, and then update $\theta := \theta \cup \{x_{k_i} \mapsto t'_{k_i}\}$. After all variables x_{k_i} were processed, apply the resulting substitution θ to $\varphi[x_1, \ldots, x_n, \overline{y}]$.

As a final remark, the applicability of DER can be increased using theory solvers. The idea is to rewrite inequalities of the form $t_1[x, \overline{y}] \neq t_2[x, \overline{y}]$, containing a universal variable x, into $x \neq t'[\overline{y}]$. This rewriting step is essentially equivalent to a theory solving step, where $t_1[x, \overline{y}] = t_2[x, \overline{y}]$ is solved for x. For example, in the case of linear bit-vector equations, this can be achieved whenever the coefficient of x is odd [44]. This observation follows from a basic result from number theory that a number a has a multiplicative inverse mod m iff $gcd(a, m) = 1$. For instance, consider the literal $3x + 2y \neq 0$, where x and y are bit-vectors of size 3. The DER rule does not apply here, since the literal is not of the form $x \neq t$. Using a theory solver for the bit-vector theory, the multiplicative inverse of $3 \ mod \ 8 (= 2^3)$ is found to be 3 as well. Therefore, $3x + 2y \neq 0$ can be rewritten into $x \neq 6y$, such that DER applies.

6.2.6 Rewriting

The idea of using rewriting for equational reasoning is not new. It traces its roots back to the work developed in the context of Knuth-Bendix completion [66]. The basic idea is to use unit clauses of the form $\forall \overline{x}.\ t[\overline{x}] = r[\overline{x}]$ as rewrite rules $t[\overline{x}] \rightsquigarrow r[\overline{x}]$, when $t[\overline{x}]$ is "bigger than" $r[\overline{x}]$. Any instance $t[\overline{s}]$ of $t[\overline{x}]$ is then replaced by $r[\overline{s}]$. For example, in the formula

$$(\forall x.\ f(x, a) = x) \ \wedge \ f(h(b), a) \geq 0,$$

the left conjunct can be used as the rewrite rule $f(x, a) \rightsquigarrow x$. Thus, the term $f(h(b), a) \geq 0$ can be simplified to $h(b) \geq 0$, producing the new formula

$$(\forall x.\ f(x, a) = x) \ \wedge \ h(b) \geq 0 \ .$$

Problems that stem from hardware as well as software verification problems often contain many equivalences of this kind. Simplification based on rewriting is therefore a promising pre-processing step which may simplify the input considerably. The goal is *not* to use rewriting to solve formulas, but to use it as an incomplete simplification technique. Therefore, there is no need to compute critical pairs or to generate a confluent rewrite system. First-order theorem provers use sophisticated *term orderings* to orient the equations $t[\overline{x}] = r[\overline{x}]$, i.e., to define what "bigger than" means (see, e.g., [54]). For the purpose of simplification, any term ordering where interpreted symbols (e.g., $+$, $*$) are considered "small", allows for simplifications in practice. This can be realized, for instance, using a Knuth-Bendix ordering where the weight of interpreted symbols is set to zero. The basic idea behind this heuristic is to replace uninterpreted symbols with interpreted ones. For example, using $f(x) \rightsquigarrow 2x + 1$, the term $f(a) - a$ may be simplified to $2a + 1 - a$. Using bit-vector rewriting rules this may be further reduced to $a + 1$.

6.2.7 Macros & Quasi-Macros

A *macro* is a unit clause of the form $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$, where f does not occur in t. Macros can be eliminated from QBVF formulas by simply replacing any term of the form $f(\overline{r})$ with $t[\overline{r}]$. Any model for the resultant formula can be extended to a model that also satisfies $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$. The following example illustrates this:

Example 10. *Consider the formula*

$$(\forall x.\ f(x) = x + a) \wedge f(b) > b\ .$$

After macro expansion, this formula is reduced to the equisatisfiable formula $b + a > b$*. The interpretation* $a \mapsto 1,\ b \mapsto 0$ *is a model for the simplified formula that can be extended to*

$$f(x) \mapsto x + 1,\ a \mapsto 1,\ b \mapsto 0\ ,$$

which is a model for the original formula.

A *quasi-macro* is a unit clause of the form

$$\forall \overline{x}. f(t_1[\overline{x}], \ldots, t_m[\overline{x}]) = r[\overline{x}] \ ,$$

where f does not occur in $r[\overline{x}]$, where $f(t_1[\overline{x}], \ldots, t_m[\overline{x}])$ contains all $\overline{x}$ variables, and the following system of equations can be solved for $x_1, \ldots, x_n$:

$$\begin{aligned} y_1 &= t_1[\overline{x}] \\ &\ldots \\ y_m &= t_m[\overline{x}] \ , \end{aligned}$$

where $y_1, \ldots, y_m$ are new variables. A solution for this system is a substitution θ of the form

$$\begin{aligned} x_1 &\mapsto s_1[\overline{y}] \\ &\ldots \\ x_n &\mapsto s_n[\overline{y}] \ . \end{aligned}$$

Let $\varphi \downarrow \theta$ denote the application of the substitution θ to the formula φ. Then the *quasi-macro* can be replaced with the *macro*

$$\forall \overline{y}. f(\overline{y}) = ite(\bigwedge_i y_i = t_i[\overline{x}], r[\overline{x}], f'(\overline{y})) \downarrow \theta$$

where ite is an if-then-else term and f' is a fresh function symbol. Intuitively, the new formula expresses that when the arguments of f are of the form $t_i[\overline{x}]$, then the result should be $r[\overline{x}]$, otherwise the value is not specified. Thereby, the quasi-macro was transformed into a macro and normal macro expansion may be applied.

Example 11. *The unit-clause*

$$\forall x. f(x+1, x-1) = x$$

is a quasi-macro, because the system $y_1 = x + 1,\ y_2 = x - 1$ *can be solved for* x. *A possible solution is the substitution* $\theta = \{x \mapsto y_1 - 1\}$. *Thus, the quasi-macro may be transformed into the macro*

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_1 = x + 1 \wedge y_2 = x - 1, x, f'(y_1, y_2)) \downarrow \theta$$

The resulting formula after application of the substitution θ *and simplifying is*

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_2 = y_1 - 2,\ y_1 - 1,\ f'(y_1, y_2)) \ .$$

Quasi-macros may not be helpful in all situations, because they require a system of equations to be solved, and the replacement terms inserted by macro expansion may not be simpler to solve than the original terms. However, in preliminary experiments on hardware verification benchmarks it was found that the system of equations was often trivially satisfied, because all variables $\overline{x}$ are actual arguments of f. For instance, assume that variable x_i is the k_i-th argument of f. Then, the substitution θ is of the form

$$\{x_1 \mapsto y_{k_1}, \ldots, x_n \mapsto y_{k_n}\} \ .$$

E.g., in many benchmarks quasi-macros of the form

$$\forall x_1, x_2.\ f(x_1,\ x_1 + x_2,\ x_2) = r[x_1, x_2]$$

(and larger versions thereof) were found.

6.2.8 Function Argument Discrimination (FAD)

After applying DER, the i-th argument of many function applications is often a bit-vector value such as: $0,\ 1,\ 2$, etc. For any function symbol f and QBV formula φ, the following macro can be conjoined with φ while preserving satisfiability:

$$\forall x, \overline{y}.\ f(x, \overline{y}) = ite(x = v, f_v(\overline{y}), f'(x, \overline{y})) \ ,$$

where f_v and f' are fresh function symbols, and v is a bit-vector value. Now, suppose that the first argument of all f-applications are bit-vector values. Macro expansion will reduce $f(v', \bar{t})$ to $f_v(\bar{t})$ when $v = v'$, and $f'(v', \bar{t})$ otherwise. The transformation can be applied again to the f' applications if their first argument is again a bit-vector value.

Example 12. *Let φ be the formula*

$$\begin{aligned} \forall x.\ & f(1, x, 0) \geq x \wedge \\ & f(0, a, 1) < f(1, b, 0) \wedge \\ & f(0, c, 1) = 0 \wedge \\ & c = a\ . \end{aligned}$$

Applying FAD twice (for the values 0 and 1) on the first argument of f results in

$$\begin{aligned} \forall x.\ & f_1(x, 0) \geq x \wedge \\ & f_0(a, 1) < f_1(b, 0) \wedge \\ & f_0(c, 1) = 0 \wedge \\ & c = a\ . \end{aligned}$$

Applying FAD for the third argument of f_1 and f_0 further reduces the formula to

$$\begin{aligned} \forall x.\ & f_{1,0}(x) \geq x \wedge \\ & f_{0,1}(a) < f_{1,0}(b) \wedge \\ & f_{0,1}(c) = 0 \wedge \\ & c = a\ , \end{aligned}$$

where all function applications have only a single argument.

While not necessarily useful in all applications, this type of simplification has a major impact on the performance of algorithms that attempt to construct (explicitly) models for uninterpreted functions. Since FAD is based on macro definitions, the infrastructure used for constructing models for macros may be used to build a model for f based on the intermediate interpretations (e.g., $f_{1,0}$ and $f_{0,1}$ in Example 12).

6.2.9 Other simplifications

As with many other SMT solvers for the bit-vector theory (e.g., [8, 19, 20]), the implementation used to evaluate the algorithms suggested in this chapter implements several bit-vector specific rewriting and simplification rules such as: $a - a \Longrightarrow 0$. These rules proved to be very effective in solving quantifier-free bit-vector formulas and they also apply in the quantified case.

Note that the following section assumes a procedure `Simplify` that, given a QBV formula φ, converts it into negation normal form, applies miniscoping, skolemization, and the other simplifications described in this section up to saturation, i.e., until no further simplification is possible.

6.3 Model Checking QBVF

Given a structure (model) M, it is useful to have a *model checking procedure* `MC` that checks whether M satisfies a universally quantified formula φ or not. To this end, a more formal definition of a model for QBVF is required.

Let BV denote the structure that assigns the usual interpretation to the (interpreted) symbols of the bit-vector theory (e.g., $+$, $*$, $concat$, etc) and let $|BV|_n$ denote the (partial) interpretation of the sort of bit-vectors of size n. With a small abuse of notation, the elements of $|BV|_n$ are $\{0_n, 1_n, \ldots, 2_n^{n-1}\}$. (Where there is no confusion, the subscript is omitted.) The interpretation of an arbitrary term t in a structure M is denoted by $M[\![t]\!]$, and is defined in the standard way. Let $M\{x \mapsto v\}$ denote a structure where the variable x is interpreted as the value v, and all other variables, function and predicate symbols have the same interpretation as in M. That is, $M\{x \mapsto v\}(x) = v$. For example, $BV\{x \mapsto 1\}[\![2 * x + 1]\!] = 3$. As usual, $M\{\overline{x} \mapsto \overline{v}\}$ denotes $M\{x_1 \mapsto v_1\}\{x_2 \mapsto v_2\} \ldots \{x_n \mapsto v_n\}$.

For each uninterpreted constant c that is a bit-vector of size n, the interpretation $M(c)$ is an element of $|BV|_n$. For each uninterpreted function (predicate) $f\colon n_1, \ldots, n_k \to n_r$ of arity k, the interpretation $M(f)$ is a term $t_f[x_1, \ldots, x_k]$, which contains only interpreted symbols and the free variables $x_1 : n_1, \ldots, x_k : n_k$. The interpretation $M(f)$ can be viewed as a *function definition*, where for all $\overline{v}$ in $|BV|_{n_1} \times \ldots \times |BV|_{n_k}$,

$$M(f)(\overline{v}) = BV\{\overline{x} \mapsto \overline{v}\}[\![t_f[\overline{x}]]\!] \; .$$

Example 13. *Let φ be the following formula:*

$$\begin{aligned}
&\forall x.\ \neg(x \geq 0) \vee f(x) < x \ \wedge \\
&\forall x.\ \neg(x < 0) \vee f(x) > x + 1 \ \wedge \\
&\qquad f(a) > b \ \wedge \\
&\qquad b > a + 1 \; .
\end{aligned}$$

Then the interpretation

$$M := \{f(x) \mapsto ite(x \geq 0, x - 1, x + 3),\ a \mapsto -1,\ b \mapsto 1\}$$

is a model for φ. For instance, $M(f(a)) = 2$.

Usually, SMT solvers represent the interpretation of uninterpreted function symbols as finite *function graphs* (i.e., lookup tables). A function graph is an explicit representation that shows the value of the function for a finite (and relatively small) number of points. For example, let the function graph $\{0 \mapsto 1,\ 2 \mapsto 3,\ else \mapsto 4\}$ be the interpretation of a function symbol g. It states that the value of the function g at 0 is 1, at 2 it is 3, and for all other values it is 4. Any function graph can be encoded using ite terms. For example, the function graph above can be encoded as $g(x) \mapsto ite(x = 0, 1, ite(x =$

$2, 3, 4))$. In practice, it is advisable to encode interpretations *symbolically* to allow for a (potential) exponentially more succinct representation. For example, assuming f is a function from bit-vectors of size 32, the interpretation $f(x) \mapsto ite(x \geq 0, x-1, x+3)$ would correspond to a very large function graph if represented explicitly, i.e., as

$$\begin{aligned} f(x) \mapsto & ite(x = 0, -1, \\ & ite(x = 1, 0, \\ & \dots \\ &)) \end{aligned}$$

When models are encoded in this fashion, it is straightforward to check whether a universally quantified formula $\forall \overline{x}.\ \varphi[\overline{x}]$ is satisfied by a structure M (see also [46]): Let $\varphi^M[\overline{x}]$ be the formula obtained from $\varphi[\overline{x}]$ by replacing any term $f(\overline{r})$ with $M[\![f(\overline{r})]\!]$, for every uninterpreted function symbol f. A structure M satisfies $\forall \overline{x}.\ \varphi[\overline{x}]$ if and only if $\neg\varphi^M[\overline{s}]$ is unsatisfiable, where $\overline{s}$ is a tuple of fresh constant symbols.

Example 14. *For instance, in Example 13, the structure M satisfies*

$$\forall x.\ \neg(x \geq 0) \vee f(x) < x \tag{6.1}$$

because

$$s \geq 0 \wedge \neg(ite(s \geq 0, s-1, s+3) < s)$$

is unsatisfiable. Let M' be a structure identical to M in Example 13, but where the interpretation $M'(f)$ of f is $x+2$. The new interpretation M' does not satisfy Equation 6.1 in φ because

$$s \geq 0 \wedge \neg(s + 2 < s)$$

is satisfiable, e.g., by $s \mapsto 0$. The assignment $s \mapsto 0$ is a counter-example *for M' being a model for φ.*

The model-checking procedure `MC` expects two arguments: a universally quantified formula $\forall\overline{x}.\ \varphi[\overline{x}]$ and a structure M. It returns $\top$ if the structure satisfies $\forall\overline{x}.\ \varphi[\overline{x}]$, and a non-empty but finite set V of counter-examples otherwise. Each counter-example is a tuple of bit-vector values $\overline{v}$ such that $M\{\overline{x} \mapsto \overline{v}\}[\![\varphi[\overline{x}]]\!] \equiv \bot$.

Using the procedure `MC`, it is possible to construct a decision procedure for QBV formulas based on iterative refinement of an initial model (Algorithm 4). This procedure starts with an initial model, which is either found heuristically or simply sets all uninterpreted functions to a fixed value. It then checks whether this model satisfies φ. If this is not the case, a counter-example V is obtained, which contains explicit values for $\overline{x}$, i.e., for all universally quantified variables. These values give a specific function point (or multiple points) at which the current model M does not satisfy φ. Instantiating the original formula with these values and solving the resulting formula with an SMT solver subsequently yields values for all the functions at the function point(s) given by V. This requires a solver for the theory of uninterpreted functions, which is a standard component in most SMT solvers and which is usually quite efficient. The function values are then used to obtain a new model M by turning all function definitions in the previous model into if-then-else terms which return the newly found value in case the parameters of the function ($\overline{p}$) match $\overline{x}$ and the previous model if they do not match $\overline{x}$.

6.4 Template Based Model Finding

In principle, the verification and synthesis problems described in section 6.2 can be attacked by any SMT solver that supports universally quantified formulas, and that is capable of producing models. Unfortunately, state-of-the-art SMT solvers do not support complete treatment of universally quantified formulas, even if the variables range over finite domains such as bit-vectors. On satisfiable instances, they will often not terminate or give up. On some un-

```
input  : QBVF $\varphi$
output: $\bot$ or a model $M$
$\varphi$ := Simplify($\varphi$) ; /* $\varphi$ is now of the form $\forall\overline{x}.\ \phi[\overline{x}]$) */
$M$ := InitialModel($\varphi$);
while True do
    $V$ := MC($\varphi, M$);
    if $V = \top$ then
        return $M$; /* valid model found. */

    $F$ := SMT_UF($\varphi\{\overline{x} \mapsto \overline{v}\}$);
    if $F = \bot$ then
        return $\bot$; /* no model exists. */
    else
        $M$ := $\{ITE(\overline{p} = \overline{x}, F(f(\overline{p})), M(f(\overline{p})))| f(\overline{p}) \in M\}$;
    end
end
```

Algorithm 4: A decision procedure for QBVF based on model refinement.

satisfiable instances, SMT solvers may terminate using techniques based on *heuristic-quantifier instantiation* [37].

It is not surprising that standard SMT solvers cannot handle these problems; the search space is simply too large. Some synthesis tools based on automated reasoning eliminate this problem (in practice) by constraining the search space using *templates*. For example, while searching for a ranking function, the synthesis tool may limit the search to functions that are linear combinations of the input before searching for more complex functions. This simple idea immediately transfers to QBVF solvers. In the context of a QBVF solver, a template is simply an expression $t[\overline{x}, \overline{c}]$ containing free variables $\overline{x}$, interpreted symbols, and fresh constants $\overline{c}$. Given a tuple of bit-vector values $\overline{v}$, we say $t[\overline{x}, \overline{v}]$ is an *instance* of the template $t[\overline{x}, \overline{c}]$. A template can also be viewed as a *parametric* function definition. For example, the template $ax + b$, where a and b are fresh constants, may be used to guide the search for an interpretation for unary function symbols. The expressions $x+1$ ($a \mapsto 1, b \mapsto 1$) and $2x$ ($a \mapsto 2, b \mapsto 0$) are instances of this template.

A *template binding* for a formula φ is a mapping from uninterpreted function (predicate) symbols f_i, occurring in φ, to templates $t_i[\overline{x}, \overline{c}]$. Conceptually, one template per uninterpreted symbol is enough. If two different templates $t_1[\overline{x}, \overline{c_1}]$ and $t_2[\overline{x}, \overline{c_2}]$ should be considered for an uninterpreted symbol f, they may simply be combined in a single new template $t'[\overline{x}, (\overline{c_1}, \overline{c_2}, c)] \equiv ite(c = 1, t_1[\overline{x}, \overline{c_1}], t_2[\overline{x}, \overline{c_2}])$, where c is a fresh constant. This approach can be extended to the construction of templates that are combinations of smaller "instructions" which can be combined to construct a template for any desired class of functions.

Without loss of generality, assume that φ contains only one uninterpreted function symbol f. A *template-based model finder* or TMF is a procedure that, given a ground formula φ and a template binding $\mathtt{TB} = \{f \mapsto t[\overline{x}, \overline{c}]\}$, returns a structure M for φ s.t. the interpretation of f is $t[\overline{x}, \overline{v}]$ for some bit-vector tuple $\overline{v}$, if such a structure exists. TMF returns $\perp$ otherwise. Assuming that φ is a ground formula enables the use of a standard SMT solver to implement TMF. It is simply the formula

$$\varphi \wedge \bigwedge_{f(\overline{r}) \in \varphi} f(\overline{r}) = t[\overline{r}, \overline{c}]$$

that needs to be checked for satisfiability. If this is the case, the model produced by the SMT solver will assign values to the fresh constants $\overline{c}$ in the template $t[\overline{x}, \overline{c}]$. Note that, when $\mathtt{TMF}(\varphi, \mathtt{TB})$ succeeds, φ is called satisfiable *modulo* TB.

Example 15. *Let φ be the formula*

$$\begin{aligned} f(a_1) &\geq 10 \wedge \\ f(a_2) &\geq 100 \wedge \\ f(a_3) &\geq 1000 \wedge \\ a_1 &= 0 \wedge \\ a_2 &= 1 \wedge \\ a_3 &= 2 \end{aligned}$$

and let the template binding TB *be $\{f \mapsto c_1 x + c_2\}$. Then, the corresponding satisfiability query is*

$$\begin{aligned} f(a_1) \geq 10 \wedge \\ f(a_2) \geq 100 \wedge \\ f(a_3) \geq 1000 \wedge \\ a_1 = 0 \wedge a_2 = 1 \wedge \\ a_3 = 2 \wedge \\ f(a_1) = c_1 a_1 + c_2 \wedge \\ f(a_2) = c_1 a_2 + c_2 \wedge \\ f(a_3) = c_1 a_3 + c_2 \ . \end{aligned}$$

The formula above is satisfiable, e.g., by the assignment $c_1 \mapsto 1$ and $c_2 \mapsto 1000$. Therefore, φ is satisfiable modulo TB*.*

6.5 A decision procedure for QBVF

The techniques described in the previous sections can be combined to produce a simple and effective solver for non-trivial benchmarks, as presented in Algorithm 5. The solver implements a form of *counter-example guided refinement* where a failed Model Checking step suggests new instances for the universally quantified formula. This method is also a variation of *model-based quantifier instantiation* [46] based on templates. The procedure `SMT` is an SMT solver for the quantifier-free bit-vector and uninterpreted function theory (QF_UFBV in SMT-LIB [7]). The procedure `HeuristicInstantiation` creates an initial set of ground instances of φ using heuristic instantiation. Note that the formula ρ is monotonically increasing in size, so the procedures `SMT` and `TMF` can exploit incremental solving features available in state-of-the-art SMT solvers.

```
input  : $\varphi$, TB
output: $\bot$ or a model $M$
1  $\varphi$ := Simplify($\varphi$) ; /* $\varphi$ is now of the form $\forall\overline{x}.\ \phi[\overline{x}]$) */
2  $\rho$ := HeuristicInstantiation($\varphi$);
3  while True do
4      if SMT($\rho$) = unsat then
5          return $\bot$;
6      $M$ := TMF($\rho$, TB);
7      if $M = \bot$ then
8          return $\bot$; /* unsat modulo TB */
9      $V$ := MC($\varphi$, $M$);
10     if $V = \top$ then
11         return $M$; /* valid model found. */
12     $\rho$ := $\rho \wedge \bigwedge_{\overline{v}\in V} \phi[\overline{v}]$ ; /* update $\rho$. */
13 end
```

Algorithm 5: A template-based QBVF decision procedure.

Theorem 8. *Algorithm 5 is complete modulo the given template* TB*.*

Proof. The formula ρ increases monotonically. The conjunct added in every iteration is an instance of φ with all universals replaced by values from the counter-example V, thereby adding new quantifier instances to ρ in every iteration. Since the number of possible instantiations is finite, the process must terminate eventually. In the case where it terminates at line 8 (with unsat modulo TB), there is no instance of the template TB that satisfies ρ. Since ρ is a conjunction of instances of φ, there is no model for φ modulo TB. $\square$

Algorithm 5 is complete for QBVF (in general) if TMF never fails, that is, M is never $\perp$. This can be accomplished using a template that simply covers *all* possible functions: Assume w.l.o.g that every function in φ has only one argument and it is a bit-vector of size 2^n. Then, using the template

$$ite(x = c_1, a_1, \ldots, ite(x = c_{2^n-1}, a_{2^n-1}, a_{2^n}) \ldots)$$

guarantees that TMF will never fail, where c_1, $\ldots c_{2^n-1}$, a_1, $\ldots$, a_{2^n} are the template parameters. Of course, it is impractical to use this template in practice. For example, the implementation used to evaluate Algorithm 5 (see Chapter 7) contains an outer-loop that increases the size of the templates whenever the Algorithm 5 returns unsat modulo TB.

In many cases, using actual tuples of bit-vector values is not the best strategy for instantiating quantifiers as shown by the following example:

Example 16. *Let f be a function from bit-vectors of size 32 to bit-vectors of the same size in*

$$(\forall x.\ f(x) \geq 0),\ \ f(a) < 0\ .$$

To prove this formula unsatisfiable, the quantifier should be instantiated with a instead of the 2^{32} possible bit-vector values. This problem may be approached as it is in [46]: Given a tuple $(v_1, \ldots, v_n)$ in V, if there is a term t in ρ s.t. $M[\![t]\!] = v_i$, use t instead of v_i to instantiate the quantifier. Of course, in practice, there may be several different t's to chose from. In this case, the syntactically smallest instance is selected, and ties are broken non-deterministically.

Additional Techniques. Templates may be used to eliminate uninterpreted function (predicate) symbols from QBV formulas. The idea is to replace any function application $f_i(\overline{r})$ (ground or not) in a QBVF φ with the template definition $t_i[\overline{r}, \overline{c}]$. The resultant formula φ' contains only uninterpreted constants and interpreted bit-vector operators. Therefore, φ' may be flattened into QBF.

Example 17. *Expanding the template $c_1x + c_2$ for f in the formula φ from Example 13, produces*

$$\begin{aligned}
&\forall x.\, \neg(x \geq 0) \vee c_1x + c_2 < x \,\wedge \\
&\forall x.\, \neg(x < 0) \vee c_1x + c_2 > x + 1 \,\wedge \\
&\quad c_1a + c_2 > b \,\wedge \\
&\quad b > a + 1 \;,
\end{aligned}$$

which readily translates to an equi-satisfiable QBF.

This observation also suggests that template model finding is essentially approximating a NEXPTIME-complete problem (QBVF) as a PSPACE-complete one (QBF). Of course, the reduction is effective if and only if the size of the templates is polynomially bounded by the size of the input formula.

If the QBV formula is a conjunction of many universally quantified formulas, a more attractive approach is quantifier elimination using BDDs [9] or resolution and expansion [14]. Each universally quantified clause can be processed independently and the resultant formulas are combined. Another possibility is to apply this approach only to a selected subset of the universally quantified sub-formulas, and rely on Algorithm 5 for the remaining cases.

Finally, first-order resolution and subsumption can also be used to derive new implied QBV universally quantified clauses and to delete those which are redundant.

6.6 Discussion & Related Work

In practice, uninterpreted functions are often not required. In this case, QBVFs can be flattened into either a propositional formula or a quantified Boolean formula (QBF). This is possible because bit-vector variables may be treated as a vector of Boolean variables. Operations on bit-vectors may be flattened to a bitwise construction of the operator. For example, bit-vector addition may be represented by a Ripple-Carry-Adder circuit on Boolean variables. This procedure is commonly referred to as *bit-blasting* and increases the size of the formula considerably (e.g., quadratically for multiplication operators), and structural information is lost.

For quantified formulas, universal quantifiers can be eliminated by expansion since each quantifies over a finite domain of values. This usually results in an exponential increase of the formula size and is therefore infeasible in practice. An alternative method is to flatten the QBV formula without expanding the quantifiers. This results in a QBF and off-the-shelf decision procedures (QBF solvers) like sKizzo [12], Quantor [14] or QuBE [50] may be employed to decide the formula. In practice, however, the performance of QBF solvers has proven to be problematic.

One of the potential issues resulting in poor performance may be the prenex clausal form of QBFs. It has thus been proposed to use non-prenex non-clausal form [2, 41, 51]. This has been demonstrated to be beneficial on certain types of formulas, but all known decision procedures fail to exploit any form of word-level (bit-vector) information.

A further problem with QBF solvers is that only few of them support certification, especially the construction of models for satisfiable instances. This is an absolute necessity for solvers employed in a synthesis context, i.e., when a concrete model is required. However, there is no standard format for QBF models and the definition of such has proven to be much less straight-forward than for SAT. (See Chapter 5 and [11, 62].)

SMT QF_BV solvers. For some time now, SMT solvers for the quantifier-free fragment of bit-vector logic (QF_BV) existed. Usually, those solvers are based on a small set of word-level simplifications and subsequent flattening (bit-blasting) to propositional formulas. Some solvers (e.g., SWORD [100]), try to incorporate word-level information while solving the flattened formula. Some tools also have limited support for quantifiers (e.g. BAT [72]), but this is usually restricted to either a single quantifier or a single alternation of quantifiers which may be expanded at feasible cost. Most SMT QF_BV solvers support heuristic instantiation of quantifiers based on E-matching [37]. On some unsatisfiable instances, this may terminate with a conclusive result, but it is of course not a solution to the general problem. The method proposed here, uses SMT solvers for the quantifier-free fragment to decide intermediate formulas, and therefore represents an extension of SMT techniques to the more general QBV logic.

Synthesis tools. Finally, there is recent and active interest in using modern SMT solvers in the context of synthesis of inductive loop invariants [91] and program fragments [59], such as sorting, matrix multiplication, de-compression, graph, and bit-manipulating algorithms. These applications share a common trait in the way they use their underlying symbolic solver. They search a template *vocabulary* of instructions, that are composed as a model in a satisfying assignment. This is the main inspiration for the template based model finding approach described in Section 6.4.

Chapter 7

Experimental Evaluation

This chapter presents an experimental evaluation of each method presented in this dissertation. When combined, these methods result in a highly efficient termination proving algorithm (Compositional Termination Analysis, Chapter 3). The methods presented in Chapters 4, 5, and 6 represent solutions to sub-problems of termination analysis and are therefore evaluated against other methods from the respective areas. The performance of each of the algorithms described in this dissertation is, in practice, highly dependent on their implementation. Each of the following sections therefore describes crucial implementation details before presenting the corresponding experiments and their interpretation. Where not otherwise stated, all experiments were executed on Intel Xeon 3 GHz machines with 16 GB of RAM.

7.1 Compositional Termination Analysis

Compositional Termination Analysis (CTA), i.e., Algorithm 2 on page 29, is implemented for analysis of ANSI-C programs. The implementation is based on the CBMC framework [26].

It instruments the program for Binary Reachability Analysis, i.e., with termination assertions as described by Cook et al. [35] and subsequently applies the termination analysis once to each loop in the program. There are two additional features that need discussion, namely an abstracting loop slicer, and the blockwise ranking procedure.

7.1.1 Slicing and Loop Abstraction

To reduce the resource requirements of the Model Checker, the implementation of CTA analyzes each loop separately. It generates an inter-procedural slice [56] of the program, slicing backwards from the termination assertion. Additionally, the program is rewritten into a single-loop program, abstracting from the behavior of all other loops.

Following the hypothesis that loop termination rarely depends on complex variables that may be calculated by other loops, the slicing algorithm replaces all assignments that depend on five or more variables with non-deterministic values. Also, all loops other than the one currently being analyzed, are 'havocked': they are replaced by program fragments that assign non-deterministic values to all variables that might change during the execution of the loop.

Note that this addresses a purely practical issue: The benchmarks used in the evaluation take too long to analyze without this abstraction. However, the abstraction is almost always precise enough, i.e., only very few termination proofs are lost. Of course, exactly the same abstracted slices are used for all methods in the evaluation.

7.1.2 Blockwise Ranking

CTA requires a ranking procedure (called $rank$ in Chapter 3). This procedure may be implementing in various ways. For example, it is possible to enumerate all paths through $\bigcup_{j=0}^{i} R^j$ and to obtain a d.wf. ranking relation for every path separately. To avoid this enumeration, the symbolic execution engine of CBMC [26] is employed to find paths through the program that are not yet included in the candidate transition invariant. To this end, a temporary program is created that first initializes all variables with non-deterministic values, saves the state, and then executes the current unwinding R^i, which is loop-free. Finally, the inclusion of all loop pre- and post-states in the current candidate transition invariant (starting with the empty set) is checked.

If a counterexample is found, a path through the program is extracted from it and $rank$ is used to compute a well-founded ranking relation for it. If this succeeds, this relation is added disjunctively to the current (d.wf.) candidate transition invariant. This procedure is essentially equivalent to the application of the Terminator Algorithm to a loop-free program fragment.

The explicit check for compositionality of the candidate transition invariant can often be avoided. For example, if we find that T is composed of a single wf. ranking relation, the transition relation must trivially be well-founded as well, since it is a subset of a wf. transition invariant.

The following example demonstrates the behavior of the implementation of CTA on a small program:

Example 18. *Consider the ANSI-C program in Figure 7.1. It contains a single loop with two potential paths through its body. Figure 7.2 presents the control flow graph of this program and defines the program locations l_1 to l_9.*

CTA starts at $i = 1$ and R^1, which is equivalent to a single unwinding of the loop, i.e., a single copy of the loop body. The initial value of the entry-state abstraction X is the complete state-space S, which includes states that have the variable $debug$ set to values other than 0. The initial termination argument T is empty $(\emptyset)$.

The blockwise ranking procedure analyzes R^1 and, since T is empty, any path through the locations l_2 and l_6 violates the current termination argument. Consider the path passing through locations $l_2, l_3, l_4, l_5, l_6, l_2$. There is

```
void main()
{
 int i;
 int debug = 0;

 while(i>0)
 {
  i--;

  if (debug)
    i=0;
 }
}
```

Figure 7.1: A program with two paths through its loop.

no well-founded ranking relation for this path because the segment between locations l_5 and l_6 fixes x to 0, i.e., x is always set to the same value, which also happens to satisfy the loop entry condition. Furthermore, the variable $debug$ never changes its value. It is therefore possible that the post-state after execution of this path is equal to the entry-state.

The ranking procedure thus returns a non-empty path precondition $C \equiv (x > 0) \wedge (debug \neq 0)$. However, C does not contain any reachable loop entry state in the original program because $debug$ is set to 0 between l_1 and l_2. Consequently, the test at line 7 of Algorithm 2 fails and the entry-state abstraction X is updated to $X \setminus C$ at line 12 of Algorithm 2.

The algorithm continues with the updated X, while T is still empty. There exist another path through the block R^1: l_2, l_3, l_4, l_6, l_2. Blockwise ranking finds a ranking function for this path, namely $-x$, and constructs the d.wf. ranking relation $T_1 \equiv -x < -x'$, which is (disjunctively) well-founded.

Finally, the current termination argument, T_1, is unified with T and i is in-

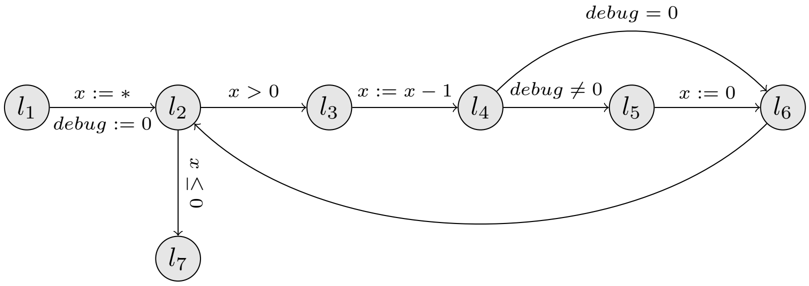


Figure 7.2: Control-flow graph of the program in Figure 7.1.

creased. Since the d.wf. ranking relation found in the previous iteration was disjunctive, compositionality of T needs to be established, which in this case is trivial. The program in Fig. 7.1 therefore terminates according to Lemma 6.

7.1.3 Experimental Results

The implementation of Compositional Termination Analysis is evaluated on a set of 87 Windows device drivers taken from the Windows Device Driver Kit (WDK).[1] The WDK already includes verification harnesses for the drivers (for verification with SDV/SLAM [5]). Every driver is analyzed in two different configurations, which results in a total of 174 benchmarks, in total containing 1665 loops. The model extractor GOTO-CC (see Appendix C) is used to extract control flow graphs from the original source files, which are then passed to the Compositional Termination Analysis engine.

CTA is compared to an implementation of the Terminator Algorithm which uses the SATABS [27] predicate abstraction engine as the safety checker. Both, the Terminator Algorithm and CTA use the simple (and incomplete) SAT-based polynomial coefficient enumeration approach for ranking relation synthesis (this method is described in Section 4.3 and evaluated in the fol-

[1] Version 6, available online at `http://www.microsoft.com/whdc/devtools/wdk/`

lowing section). All experiments are run using a timeout of one hour and a memory limit of 2 GB.

Whenever CTA is not able to find a valid d.wf. transition invariant for a loop, it returns the weakest precondition of a path through the program. This precondition describes a set of states from which termination of the program is not guaranteed. The implementation can be configured to react to this situation in three different ways:

a) check reachability of the precondition,

b) check reachability of the loop, or

c) report the loop as non-terminating.

Results are presented for all three variants.

First, the results obtained from variant a) are discussed. This variant checks path preconditions using a Model Checker (here this is SATABS). This variant is a full implementation of the CTA algorithm.

Every data point in Figure 7.3 represents one loop. On the horizontal axis the total time taken to analyze the loop using the Terminator Algorithm is indicated. The vertical axis indicates the time taken by Compositional Termination Analysis. It is apparent from Figure 7.3, Compositional Termination Analysis is up to three orders of magnitude faster. However, there are a few non-terminating benchmarks on which it is slower or runs out of memory. Due to this behavior, CTA loses precision compared to Terminator on 60 benchmarks, i.e., it reports a loop as non-terminating where Terminator proves termination.[2] This is due to the fact that on non-terminating loops many (or all) path preconditions are eventually enumerated. The resulting loop-free programs are sometimes too difficult for the model checker. A possible solution for this problem is techniques that compute a more general precondition of

[2]The implementation reports a loop as non-terminating when it runs out of resources.

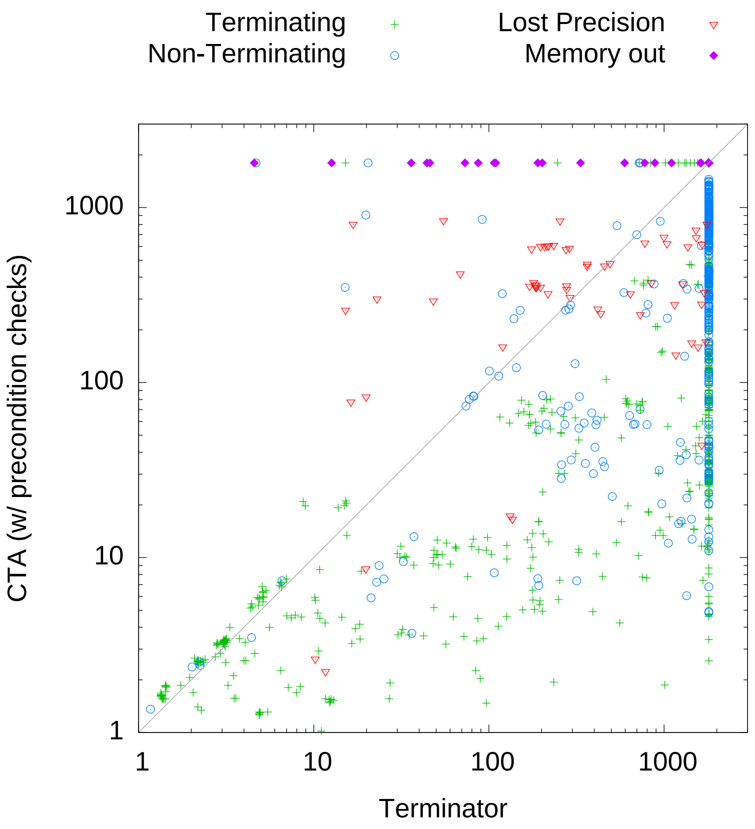


Figure 7.3: Experimental results using path precondition checks (CTA a).

non-termination. A recent technique described by Cook et al. [32], which constructs preconditions of termination, may be adapted for this purpose.

Figure 7.4 provides the results obtained when checking for general loop reachability, which is essentially a crude over-approximation of the precondition of the non-terminating paths through the loop. The results are very similar to those of the previous variant, which is due to the fact that most loops are

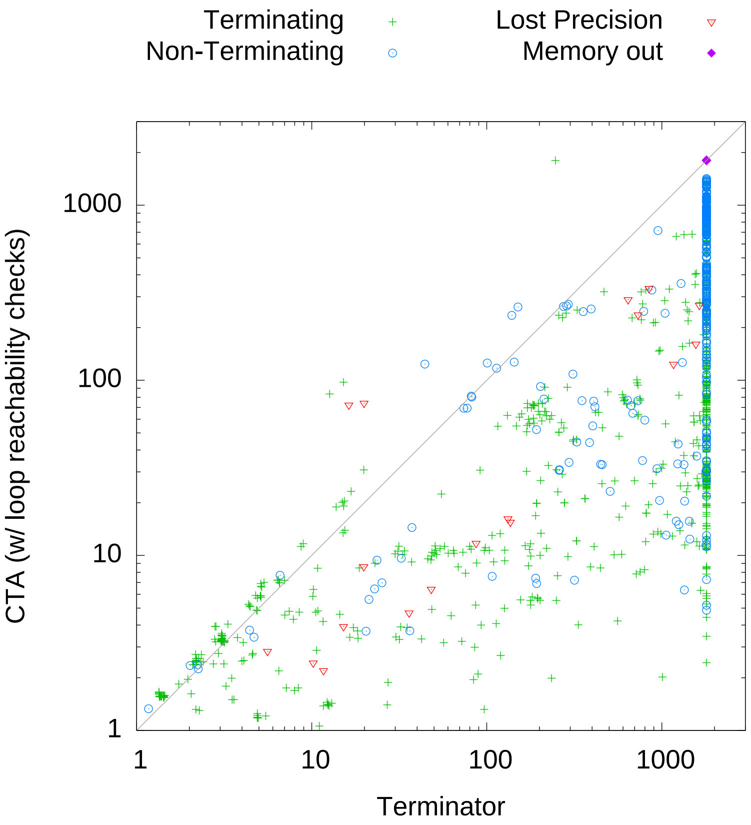


Figure 7.4: Experimental results using loop reachability checks (CTA b).

indeed reachable and so are most path preconditions. The difference in precision compared to Terminator is very small for this variant: only 18 termination proofs are lost due to the over-approximation. In terms of performance however, variant b) is far superior to Terminator. The data points in Figure 7.4 are almost all below the diagonal and many of them are in the area of two to three orders of magnitude of improvement.

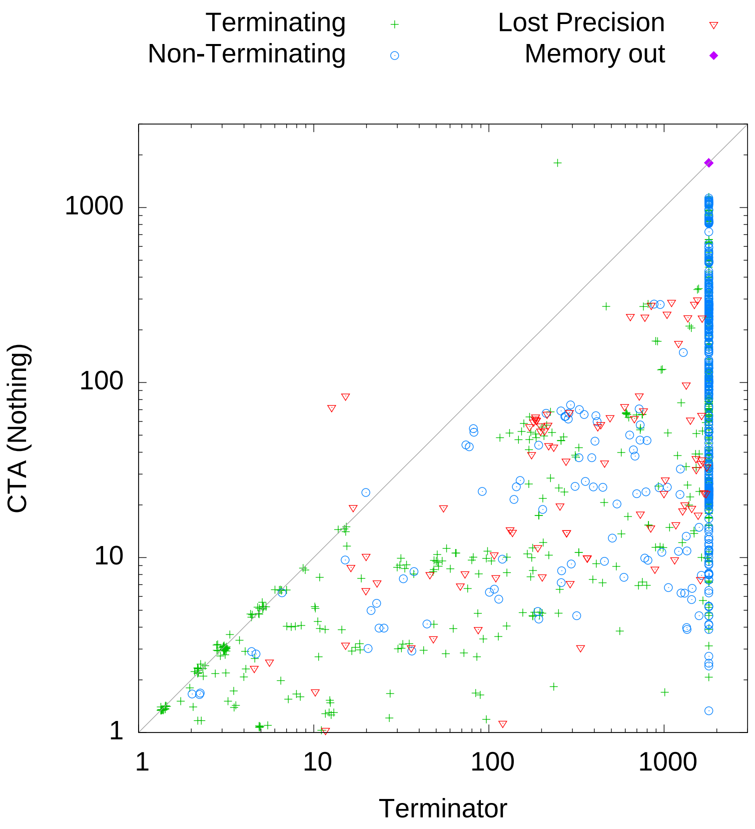


Figure 7.5: Experimental results without loop reachability or precondition checks (CTA c).

Finally, CTA variant c) reports non-termination immediately, i.e., without checking reachability of the loop or a precondition. The results obtained using this very crude approximation are presented in Figure 7.5. Naturally, this version of CTA is the fastest, but it also introduces unwanted imprecision. However, in practice it is still rare for this imprecision to manifest itself: only

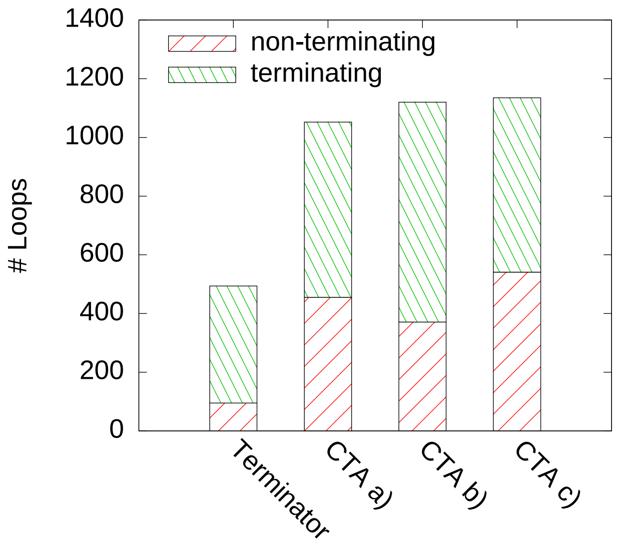


Figure 7.6: Total number of loops analyzed within the resource limits.

89 termination proofs are lost, compared to the Terminator Algorithm.

For a comparison of the total number of termination proofs obtained through each algorithm, see Figure 7.6. The overall capacity of Compositional Termination Analysis is clearly much higher than that of the Terminator Algorithm: Compositional Termination Analysis is able to analyze more than twice the number of benchmarks that Terminator is able to analyze. The data in Figure 7.6 also indicates that CTA variant b) may be a good solution in practice, since it is most successful when bounded by resource limits.

The raw experimental data obtained for this evaluation is too voluminous to be included in this dissertation. It is however, available on the web, at `http://www.cprover.org/termination/`, along with the implementation, and additional material.

7.2 Ranking Relation Synthesis

Chapter 4 presents two new methods for ranking relation synthesis for bit-vector programs. Those methods are compared to existing methods in this section.

The most important comparison is that to the linear ranking function synthesis engine RANKFINDER, which uses rational arithmetic but is otherwise very similar to the method based on integer linear programming (Section 4.2).

Finite-state programs using *only* machine integers can also be proved terminating without ranking functions. Therefore, the performance of the new methods is also compared with one approach not based on ranking functions: the rewriting of termination properties to safety properties according to Biere et al. [15]. Note that because the vast majority of systems-level code makes extensive use of both bit-level operations on machine integers and unbounded data-structures such as linked lists however, ranking function synthesis for machine integers remains advantageous.

7.2.1 Large-scale benchmarks

The termination prover used for this evaluation is the implementation of the Terminator algorithm as discussed in the previous section. The benchmarks are device drivers from the Windows Driver Development Kit (WDK), again as discussed in the previous section. The model extractor GOTO-CC (see Appendix C) is used to extract model files from a total of 87 drivers in the WDK.

Most of the drivers contain loops over singly and doubly-linked lists, which require an arithmetic abstraction. This abstraction can be automated by existing shape analysis methods (e.g., that recently presented by Yang et al. [102]). Note that the implementation used in this evaluation does not include any abstraction for this purpose.

hidusbfx2	cdrom	isousb	toaster-filter	tape	sfilter	kbdclass	disk	bulkusb	Benchmark	
2	8	34	1	6	10	13	14	13	total	Loops
2	4	0	1	3	10	7	7	13	terminating	
0	4	11	0	3	0	5	7	0	non-terminating	
2	0	7	1	1	0	2	2	6	# Rank Functions	
1	340	—	61	10	37	—	218	—	Time [min]	

Table 7.1: A small selection of the results on full driver code.

Just like Cook et al. [36], we find that most of the runtime of the Terminator algorithm is spent in the reachability checker (more than 99%), especially after all required ranking functions have been synthesised and no more counterexamples exist. To reduce the resource requirements of the Model Checker, the implementation used here analyzes each loop separately and generates an inter-procedural slice [56] of the program, slicing backwards from the termination assertion. In addition, the program is rewritten into a single-loop program, abstracting from the behavior of all other loops (see also Section 7.1.1). With this (abstracting) slicer in place, the absolute runtime and memory requirements are reduced dramatically.

A small subset of the results is presented in Table 7.1. Note that, if ranking functions are successfully synthesized, but the final safety property can not be proven within the time limit, the loop is classified as non-terminating. The entry '—' indicates that the tool ran out of time (after 6 hrs) or memory (2GB). Consequently the numbers of terminating and non-terminating loops do not necessarily add up to the total number of loops. The complete data on all drivers is voluminous; for the purpose of this discussion it is sufficient to present a typical example in detail. The full dataset is available online.[3]

[3]`http://www.cprover.org/termination/`

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**Loop**
list	list	unr.	i++	unr.	unr.	unr.	unr.	wait	unr.	unr.	i++	list	**Type**
126	85	687	248	340	298	253	844	109	375	333	3331	146	**CE Time [sec]**
0.5	0.1	–	0.7	–	–	–	–	0.4	–	–	2.2	0.4	**Synth. Time [sec]**
$\times$	$\times$	✓	MO	✓	✓	✓	✓	$\times$	✓	✓	MO	$\times$	**Terminates?**

Table 7.2: The behaviour on the loops of a keyboard driver.

The keyboard class driver in the WDK (KBDCLASS) contains a total of 13 loops in a harness (SDV_FLAT_ HARNESS) that calls all dispatch functions nondeterministically. Table 7.2 describes the behaviour of the Terminator algorithm on this driver. For every loop this table lists the type of the loop (list iteration, integer increment (i++), unreachable, or 'wait for device'), the time it takes to find a potentially non-terminating path ('CE Time'), the time required to find a ranking function using the incomplete SAT template (Equation 4.11 in Section 4.3) ('Synth. Time', where applicable), and the final outcome. In the last row, 'MO' indicates a memory-out after consuming 2 GB of RAM whilst proving that no further counterexamples to termination exist. The entire analysis of this driver requires 2 hours.

In the course of this evaluation, a possible termination problem in the USB driver bulkusb was identified, which may result in the system being blocked. In module bulkpnp of the WDK sample USB driver (bulkusb) the driver requests that an interface description structure is searched within a ConfigurationDescriptor for every device available. It increments the loop counter if this did not return an error. The function USBD_ParseConfigurationDescriptorEx, however, is an API function for which no implementation is available. According to the API documentation, it may return NULL if no interface matches the search criteria (iIndex, 0, -1, -1, -1 in Fig. 7.7), resulting in iNumber not being incremented. Since numberOfInterfaces is a local (non-shared) variable the loop, the problem would persist in a concurrent setting, where the device may be disconnected while the loop is executed.

```
while(iNumber < numberOfInterfaces) {
  iDesc = USBD_ParseConfigurationDescriptorEx(
                     ConfigurationDescriptor,
                     ConfigurationDescriptor,
                     iIndex,
                     0, -1, -1, -1);
  if(iDesc) {
    /* ... */
    iNumber++;
  }
  iIndex++;
}
```

Figure 7.7: Code fragment from `usb/bulkusb/sys/bulkpnp.c` (simplified)

7.2.2 Experiments on smaller examples

The dominance of the reachability engine in the large-scale experiments prevents a meaningful comparison of the utility of the various techniques for ranking function synthesis. For this reason, further experiments on smaller programs were conducted, where the behavior of the reachability engine has less impact. For this purpose, 61 small benchmark programs were extracted manually from the WDK drivers. Most of them contain bit-vector operations, including multiplication, and some of them contain nested loops. All benchmarks were manually sliced by removing all source code that does not affect program termination (much like an automated slicer, but more thoroughly). The same abstraction technique as described in Section 7.1.1 was used. Out of these 61 benchmarks, 10 are non-terminating. The time limit in these benchmarks was 1 hour, and the memory consumption was limited to 2 GB.

To evaluate the integer linear programming method described in Sect. 4.2,

#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Manual Insp.	L	L	L	L	N	N	N	L	T	N	T	L	L	N	T	L	L	L	L	L	T	L	L	L	L	L	L	N	T	L	T
SAT	●	●	●	●	○	○	○	●	○	○	○	●	●	○	○	●	●	●	●	●	○	●	●	●	●	●	○	○	○	●	○
Seneschal	●	●	●	●	○	○	–	●	○	○	○	●	●	○	○	●	●	●	●	●	○	–	●	●	●	●	○	○	○	–	○
Rankfinder	○	●	○	●	○	○	○	●	◐	○	○	●	○	◐	◐	●	○	●	●	○	○	○	●	●	●	●	◐	○	–	○	○
QBF [-1,+1]	–	–	●	●	○	○	–	–	–	○	–	●	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	○	–	●	○
QBF $P(c,x)$	–	–	●	●	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–
Biere et al. [15]	–	–	–	●	–	–	–	–	–	○	–	●	●	–	–	–	–	–	–	●	–	–	●	–	–	–	–	○	–	–	●

#	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	
Manual Insp.	T	L	L	N	L	T	L	L	L	L	L	L	N	T	L	L	T	T	T	L	T	T	N	L	L	L	L	L	N	T	
SAT	○	●	○	○	●	○	●	●	●	●	●	●	○	○	●	●	○	○	○	●	○	○	○	●	●	●	●	●	○	○	
Seneschal	○	●	○	○	●	○	●	●	●	●	●	●	○	○	●	●	○	○	○	●	○	○	○	●	●	●	●	–	○	–	
Rankfinder	○	●	○	○	●	◐	●	●	○	○	●	●	○	○	●	●	○	○	○	●	–	○	–	●	○	●	●	○	○	○	
QBF [-1,+1]	○	–	–	–	–	–	–	–	●	–	–	●	○	–	–	●	–	○	○	●	○	–	–	–	–	–	–	–	○	–	
QBF $P(c,x)$	–	–	–	–	–	–	–	–	–	–	–	–	○	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	
Biere et al. [15]	–	–	–	–	●	–	–	–	●	–	–	–	–	●	●	●	–	–	●	●	–	–	–	–	–	–	–	–	○	●	

●	–	Termination was proven	T	–	Terminating (non-linear)
○	–	(Possibly) Non-terminating	L	–	Terminating, and linear ranking functions exist.
◐	–	Incorrect under bit-vector semantics			
–	–	Memory or time limits exhausted	N	–	Non-terminating

Table 7.3: Experimental results on 61 benchmarks drawn from Windows device drivers.

a prototype called Seneschal[4] was developed. It is based on the theorem prover Princess [86] for Presburger arithmetic (PA) with uninterpreted predicates and works by (i) translating a given bit-vector program into a PA formula, (ii) eliminating the quantifiers in the formula, (iii) flattening the formula to a disjunction of systems of inequalities, and (iv) applying Lem. 9 to compute ranking functions. Seneschal does not currently transform systems of inequalities to integral systems, which means that it is a sound but incomplete tool; the experiments show that transformation to integral systems is unnecessary for the majority of the programs considered.

For the experiments using Rankfinder[5], the bit-vector operators $+$, $\times$ with literals, $=$, $<_s$ and $<_u$ are approximated by the corresponding operations

[4]`http://www.philipp.ruemmer.org/seneschal.shtml`
[5]`http://www.mpi-inf.mpg.de/~rybal/rankfinder/`

on the rationals, whereas nonexistence of ranking functions is reported for programs that use any other operations. Furthermore, constraints of the form $0 \leq v < 2^n$, where n is the bit-width of v, are added to restrict the range of pre-state variables.

The other methods are implemented using the CBMC framework [26] to flatten bit-vector formulas into SAT or QBF formulas (without loss of precision).

Table 7.3 summarizes the results. The first column indicates the results obtained by manual inspection, i.e., whether a specific benchmark is terminating, and if so whether there is a linear ranking function to prove this. The other columns represent the following ranking synthesis approaches: SAT is the coefficient enumeration approach from Section 4.3; Seneschal is the integer linear programming approach from Section 4.2.1; Rankfinder is the linear programming approach over rationals described in Section 2.2.3; QBF [-1,+1] is a QBF template approach from Section 4.3 with coefficients restricted to $[-1, +1]$, such that the template represents the same ranking functions as the one used for the SAT enumeration approach. QBF $P(c, x)$ is the unrestricted version of this template. Note that two benchmarks (#27 and #34) are negatively affected by the abstracting slicer: due to the abstraction, no linear ranking functions are found. On the original programs, the SAT-based approach and Seneschal find suitable ranking functions. On benchmark #34 however, the Model Checker times out afterwards.

Comparing the various techniques, it is clear that the simple SAT-based coefficient enumeration is most successful in synthesizing useful ranking functions. It is able to prove 34 out of 51 terminating benchmarks and reports 27 as non-terminating. It does not time out on any instance (with slicing; on one instance without slicing).

Seneschal shows the second best performance: it proves 31 programs as terminating, almost as many as the SAT-based template approach. It reports 25 benchmarks as non-terminating and times out on 5.

The interpretation of bit-vectors as rationals using the RANKFINDER engine results in 23 successful termination proofs, and 35 cases of alleged non-termination. In three cases, the reachability checker times out on proving the final property, and in 5 cases RANKFINDER returns an unsuitable ranking function.

The runtimes of the SAT coefficient enumeration, Seneschal, and Rankfinder are relatively short (seconds). Tables 7.4 and 7.5 list the runtime and the final result for all approaches and all benchmarks.

For the two QBF techniques, an experimental version of QuBE was used as the QBF solver. This solver performed better than sKizzo, Quantor, and Squolem in a preliminary evaluation. The constrained template ($QBF[-1,+1]$) is still able to synthesize some useful ranking functions within the time limit. It proves 9 benchmarks terminating and reports 11 as non-terminating. The unconstrained approach (QBF $P(c,x)$), however, proves only 5 programs terminating and one non-terminating, with the QBF-Solver timing out on all other benchmarks.

Since all programs in the benchmark set are finite-state programs, it is necessary to compare to existing methods which do not require ranking functions. Here, they are compared to an approach suggested by Biere et al. [15] (bottom row of Table 7.3). This approach does not require ranking functions, but instead proves that an entry state of the loop is never revisited. Generally, these assertions are difficult for the reachability checker (here SATABS). While this method is able to show only 14 programs terminating, there are 4 benchmarks (#31, #45, #50, and #61) that none of the other methods can handle as they require non-linear ranking functions.

#	Manual	SAT		Seneschal		Rankfinder		QBF [-1,+1]		QBF $P(C, \mathcal{X})$		Biere et al. [15]	
1	L	52.07	●	17.67	●	0.05	○	T/O	–	T/O	–	T/O	–
2	L	1.16	●	22.02	●	1.19	●	T/O	–	T/O	–	T/O	–
3	L	0.30	●	10.97	●	0.03	○	49.45	●	567.79	●	3595.58	–
4	L	0.29	●	7.60	●	0.37	●	16.28	●	220.65	●	10.95	●
5	N	0.18	○	15.15	○	0.04	○	0.78	○	T/O	–	T/O	–
6	N	0.18	○	20.33	○	0.03	○	1.57	○	T/O	–	T/O	–
7	N	17.93	○	T/O	–	0.03	○	T/O	–	T/O	–	T/O	–
8	L	0.40	●	8.36	●	0.36	●	T/O	–	T/O	–	T/O	–
9	T	0.12	○	8.05	○	0.08	○	T/O	–	T/O	–	T/O	–
10	N	0.25	○	9.62	○	0.02	○	0.78	○	T/O	–	0.04	○
11	T	0.28	○	11.54	○	0.04	○	T/O	–	T/O	–	T/O	–
12	L	0.25	●	7.57	●	0.31	●	2.45	●	18.75	●	50.32	●
13	L	0.28	●	8.68	●	0.04	○	T/O	–	T/O	–	88.27	●
14	N	0.28	○	7.88	○	0.04	○	T/O	–	T/O	–	T/O	–
15	T	0.57	○	14.82	○	0.27	○	T/O	–	T/O	–	T/O	–
16	L	1.65	●	12.12	●	0.51	●	T/O	–	T/O	–	T/O	–
17	L	1.10	●	16.86	●	0.26	○	T/O	–	T/O	–	T/O	–
18	L	9.88	●	14.54	●	0.68	●	T/O	–	T/O	–	T/O	–
19	L	0.38	●	7.47	●	0.16	●	T/O	–	T/O	–	T/O	–
20	L	0.31	●	8.56	●	0.01	○	T/O	–	T/O	–	1.09	●
21	T	8.09	○	0.07	○	0.06	○	T/O	–	T/O	–	T/O	–
22	L	0.36	●	T/O	–	0.02	○	T/O	–	T/O	–	T/O	–
23	L	0.44	●	14.09	●	0.48	●	13.81	●	570.24	●	1.09	●
24	L	0.60	●	8.36	●	0.69	●	T/O	–	T/O	–	T/O	–
25	L	0.35	●	7.64	●	0.18	●	T/O	–	T/O	–	T/O	–
26	L	0.38	●	7.70	●	0.20	●	T/O	–	T/O	–	T/O	–
27	L	1.65	○	16.36	○	0.20	○	T/O	–	T/O	–	T/O	–
28	N	0.08	○	8.95	○	0.03	○	0.24	○	T/O	–	9.02	○
29	T	0.29	○	8.15	○	T/O	–	T/O	–	T/O	–	3539.23	–
30	L	0.30	●	T/O	–	0.02	○	1735.81	●	T/O	–	T/O	–
31	T	0.10	○	23.16	○	0.03	○	0.25	○	T/O	–	2.46	●

Table 7.4: Runtime (in seconds) of various ranking synthesis techniques on 61 Windows device driver loops (Part 1).

#	Manual	SAT		Seneschal		Rankfinder		QBF [-1,+1]		QBF $P(C,\mathcal{X})$		Biere et al. [15]	
32	T	1.00	○	6.04	○	0.10	○	0.79	○	T/O	–	T/O	–
33	L	0.39	●	7.52	●	0.16	●	T/O	–	T/O	–	T/O	–
34	L	1114.95	○	217.93	○	0.05	○	T/O	–	T/O	–	T/O	–
35	N	0.36	○	16.07	○	0.39	○	T/O	–	T/O	–	T/O	–
36	L	0.32	●	7.43	●	0.20	●	T/O	–	T/O	–	1.83	●
37	T	0.80	○	14.66	○	0.54	○	T/O	–	T/O	–	T/O	–
38	L	0.35	●	7.22	●	0.38	●	T/O	–	T/O	–	3529.21	–
39	L	4.37	●	11.80	●	2.10	●	T/O	–	T/O	–	T/O	–
40	L	0.14	●	1071.52	●	0.03	○	1.26	●	T/O	–	18.39	●
41	L	0.44	●	11.00	●	0.03	○	T/O	–	T/O	–	T/O	–
42	L	0.71	●	15.09	●	0.77	●	T/O	–	T/O	–	T/O	–
43	L	2.59	●	8.00	●	2.26	●	2.96	●	T/O	–	T/O	–
44	N	0.29	○	6.76	○	0.31	○	17.43	○	572.51	○	T/O	–
45	T	0.28	○	9.62	○	0.02	○	T/O	–	T/O	–	0.55	●
46	L	0.28	●	7.31	●	0.29	●	T/O	–	T/O	–	0.19	●
47	L	0.14	●	7.77	●	0.09	●	1.37	●	T/O	–	40.43	●
48	T	0.24	○	8.44	○	0.02	○	T/O	–	T/O	–	3540.10	–
49	T	0.24	○	7.72	○	0.03	○	0.62	○	T/O	–	T/O	–
50	T	0.23	○	8.18	○	0.03	○	0.66	○	T/O	–	1310.13	●
51	L	0.46	●	13.98	●	0.47	●	21.03	●	218.50	●	4.92	●
52	T	0.24	○	7.44	○	T/O	–	1.31	○	T/O	–	T/O	–
53	L	0.30	○	3.31	○	0.07	○	T/O	–	T/O	–	3549.58	–
54	N	0.25	○	7.02	○	T/O	–	T/O	–	T/O	–	T/O	–
55	L	0.28	●	7.48	●	0.29	●	T/O	–	T/O	–	T/O	–
56	L	1.01	●	8.57	●	0.04	○	T/O	–	T/O	–	T/O	–
57	L	0.61	●	14.76	●	0.67	●	T/O	–	T/O	–	T/O	–
58	L	14.61	●	24.31	●	1.56	●	T/O	–	T/O	–	3535.98	–
59	L	0.21	●	T/O	–	0.03	○	T/O	–	T/O	–	T/O	–
60	N	0.24	○	7.75	○	0.03	○	0.74	○	T/O	–	0.04	○
61	T	6.68	○	T/O	–	0.05	○	T/O	–	T/O	–	1.88	●

Table 7.5: Runtime (in seconds) of various ranking synthesis techniques on 61 Windows device driver loops (Part 2).

7.3 Certificates for QBF

Certificate extraction for QBF as described in Chapter 5 was implemented in three different solvers:

- SQUOLEM, a *new* skolemization-based solver,
- EBDDRES, an existing BDD-based solver [63, 90], and
- QUAFFLE, an existing search-based solver [104].

All of the solvers were instrumented with support for certificate extraction from invalid formulas, i.e., they support Q-resolution proofs. Model extraction based on Skolem-functions is supported by SQUOLEM and EBDDRES.

SQUOLEM. The first of the solvers, SQUOLEM, is a *new* skolemization-based solver, which generates both models and refutations. SQUOLEM eliminates quantifiers from a QBF by explicitly generating Skolem-functions for existential variables in the inner-most scope. For each variable that is about to be eliminated, it collects all clauses in which the variable occurs and interprets them as implications, e.g., $(a \vee b)$ is interpreted as $\neg a \rightarrow b$. The set of these implications essentially forms a function definition, by which the variable is replaced.

In the case of conflicting implications, e.g., $\neg a \rightarrow b$ and $\neg a \rightarrow \neg b$, a clause stating that the conflict must be avoided, e.g., (a), is added. In terms of the (Q-)resolution calculus, the conflict clause is simply a resolvent obtained from the two conflicting implications. The process is iterated until either a complete model has been constructed, or conflicting unit clauses occur (i.e., an empty clause is resolved). In the first case, the model is written to the certificate file, in the second case a Q-resolution tree is constructed from information about a clause's parents that is recorded during model construction.

EBDDRES. EBDDRES is a BDD-based QBF solver that eliminates variables starting from the innermost scope. In order to eliminate a variable x, EBDDRES first builds a conjunction of all the clauses containing x and then quantifies x using one standard BDD OR- resp. AND-operation if the variable is existential resp. universal. Once all variables are eliminated, a constant BDD is obtained.[6]

EBDDRES produces refutations by introducing a new (Tseitin-) variable for each BDD node, which is defined to be an if-then-else gate. Given a BDD node n, let x be its variable and t, t_0, and t_1 the (Tseitin-) variables introduced for n, its left child, and its right child, respectively. Then the definition of t is

$$t \Leftrightarrow (x \; ? \; t_1 \; : \; t_0) \; .$$

Thus, if x is true (false), the Tseitin variable t_1 (t_0) has to be true. A refutation is constructed by first showing that the Tseitin variables for the root nodes of the BDDs for every original clause have to be true. The logical operations of the solving algorithm are traced until it is shown that the Tseitin variable for the constant BDD zero has to be true, an clear contradiction. The full details (with the exception of universal quantification) can be found in [63, 90]. For universal quantification, the proof rule is as follows: Let x be a universal variable to be eliminated and let t be the variable corresponding to the root node of the BDD which is the conjunction of all the clauses containing x. Clearly, t must be true for the formula to be valid. The definition of t introduces (among others) the clause $(\neg t \vee x \vee t_0)$. Resolving this with (t) yields $(x \vee t_0)$ which, based on the definition of the Quantified Extension Rule (Definition 27), can be forall-reduced to (t_0) since t_0 is not in the scope of x. The proof for t_1 is similar.

Models are produced as follows. In the bucket algorithm (which EBDDRES implements), when variable x is to be eliminated, a BDD containing precisely all the constraints on x is built. Traversing the BDD from the root to the child where x is true (right child) gives another BDD that encodes all the valuations

[6]For simplicity, the variable ordering in the BDDs is the reverse of the QBF variable order.

where x has to be true to satisfy all the constraints. If x is existential, then this is precisely a Skolem-function for x. Thus, the certificate consists of definitions of the Tseitin variables for this BDD and the Skolem-function is set to be equivalent to the Tseitin variable of the root (analogously, we could have chosen the negation of the left child).

QUAFFLE. The search-based solver QUAFFLE is, so far, limited to refutation traces and does not produce models. Unfortunately, in case of this solver, (conflict-driven) learning had to be disabled, since it uses long-distance resolution [104] and it is currently unclear whether it is possible to trace this operation in the model format proposed in Chapter 5.

Refutations are constructed as follows: Assume (without loss of generality) that the innermost scope is existential. If the instance is invalid, both choices for the truth value of an existential variable lead to a conflict. For a universal variable, at least one choice leads to a conflict. Whenever a conflict is reached, a conflict clause can be derived.

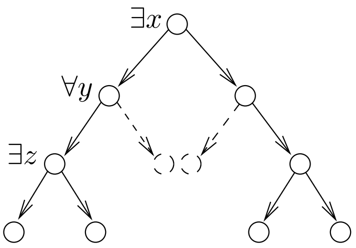


Figure 7.8: QUAFFLE search tree.

Refutation proofs are constructed from these conflict clauses. Consider for instance the simple search tree example presented in Figure 7.8 and let the solid lines denote paths to conflicts. Let the left (right) arrow from a node represent setting the truth value of a variable to false (true). Now, the leftmost path $(\neg x \wedge \neg y \wedge \neg z)$ leads to a conflict producing the conflict clause $(x \vee y \vee z)$. Similarly, the path $(\neg x \wedge \neg y \wedge z)$ produces the clause $(x \vee y \vee \neg z)$. These clauses are then resolved (and forall-reduced) to first obtain $(x \vee y)$ and then (x). Similarly, from the conflict clauses from the right-hand side (where x is true) the clause $(\neg x)$ is obtained. Resolving the two, results in an empty clause. The above representation is simplified. Prior experimental work has revealed that in most cases rather than a complete conflict clause, one that subsumes it is obtained. The consequence of this is that it is possible to omit some resolution steps.

If a formula has a large alternation depth, the proof-generation algorithm starts by resolving the literals from the innermost existential scope. Subsequently, the new clauses are forall-reduced to eliminate the enclosing universal scope. Then, the proof generation eliminates the second innermost existential scope and thus alternates between resolution and forall-reduction until the outermost scope is reached.

QBV. To verify the certificates, a tool called *Quantified Boolean Verifier* or QBVwas implemented. The algorithm underlying this tool executes applications of the extension rule and Q-resolution steps, as listed in the certificate. The last statement in a certificate is a conclusion line that either provides the index of an empty clause (for refutations), or a list of equivalences of variables for a model. As suggested by Benedetti [11], every clause is checked separately against the model. For this purpose the SAT-solver MINISAT (Version 1.14p) is used in incremental mode, i.e., the model is loaded into MINISAT and then the negation of a clause is added as an assumption, which must result in an unsatisfiable problem. As stated earlier, this problem is in general Co-NP complete.

7.3.1 Experimental Results

To show that certificate extraction is feasible, experiments were conducted on benchmarks from the 2005 QBF-Eval[7] dataset (fixed instances) and the 2006 preliminary dataset, which totals to 445 test cases. The three solvers, EBDDRES, QUAFFLE, and SQUOLEM are used to generate certificates, which are subsequently verified using QBV.

The tests for EBDDRES and QUAFFLE were run on a cluster of Intel Pentium IV PCs (3.0 GHz) with 2 GB RAM while those for SQUOLEM were run on an Intel Xeon (3.0 GHz) PC with 4 GB RAM. Each test was run using a time limit of 600 seconds and a memory limit of 1 GB.

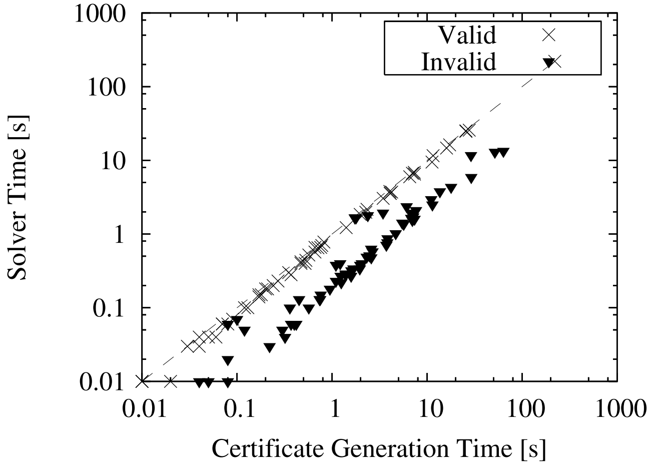
EBDDRES is able to create a model for 80 instances and a refutation trace for 86 instances. QUAFFLE is able to produce 26 refutations (without learning).

The first interesting performance figure is the time required to solve an instance (not producing a certificate) in relation to the time that it takes to generate a certificate. The results for EBDDRES and QUAFFLE are shown in Figures 7.9(a) and 7.9(b), respectively. They show that for both solvers, generating a certificate takes longer than solving the instance. For EBDDRES the overhead for models is about 15% and for refutations 440%. This behavior is (most likely) due to the fact that the Q-resolution proofs for EBDDRES are so large that merely saving them to a file takes considerable time. Note that a similar overhead is also observed in the unquantified case [63, 90].

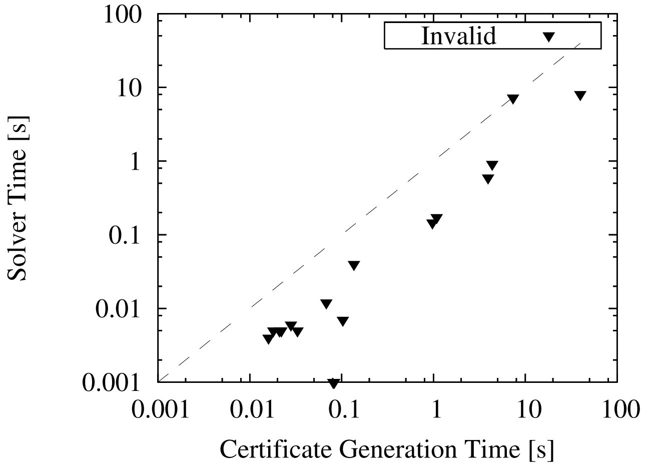
For QUAFFLE the overhead is even higher, 1440%. However, this result is not illustrative since this solver solves only very few instances and these instances belong to only 6 families. For one instance, 'k_lin_p-4', the trace generation overhead is only 1.6%.

The second performance figure to be evaluated is the time required to validate a certificate in relation to the time that it takes to generate them. The results for EBDDRES and QUAFFLE are shown in Figures 7.10(a) and 7.10(b).

[7] http://www.qbflib.org/

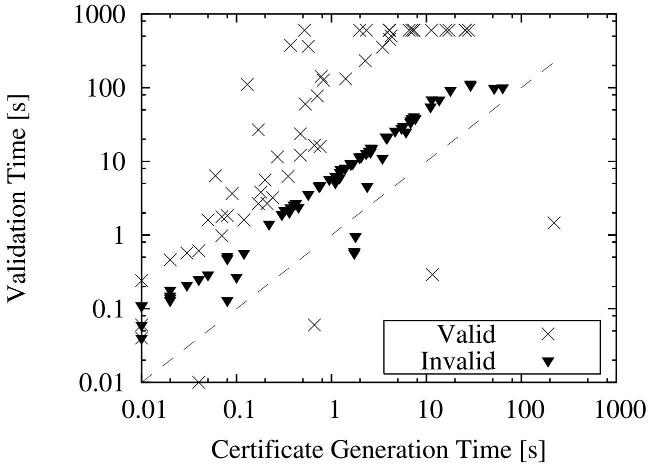


(a) EBDDRES

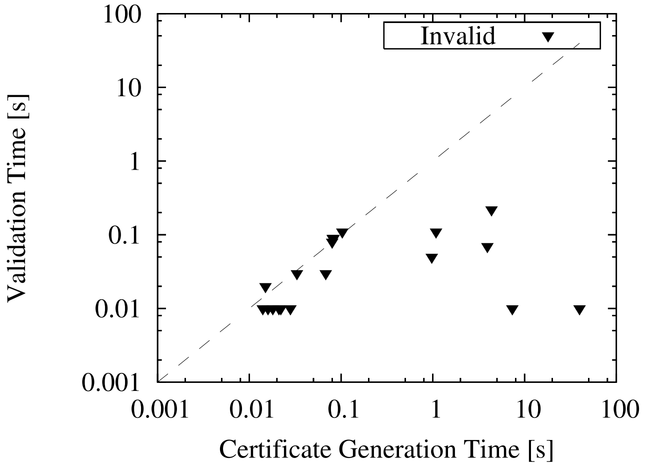


(b) QUAFFLE

Figure 7.9: Certificate Generation vs. Solver Time for EBDDRES and QUAFFLE.



(a) EBDDRES



(b) QUAFFLE

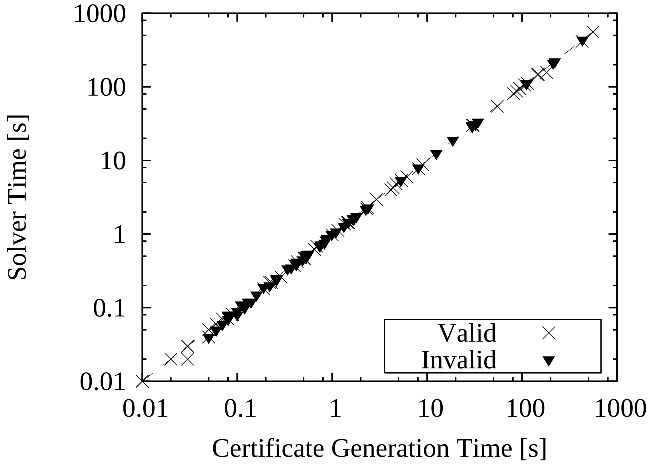
Figure 7.10: Certificate Generation vs. Validation Time EBDDRES and QUAFFLE.

These results show that, for EBDDRES, it takes on average more time to validate than it takes to produce a certificate; again the overhead depends heavily on whether the instance is valid or invalid. Validating models, takes on average over 100 times longer than certificate generation. QBV actually times out on 15 instances (after 600 seconds) where model generation is feasible. Refutations, on the other hand, are verified quicker, the ratio is 5.4. For QUAFFLE, the trace validation times are neglible (on average < 0.04 seconds), as Figure 7.10(b) shows. The behavior observed is in line with the fact that verifying a model is in general Co-NP complete whereas a resolution trace can be verified in polynomial time.

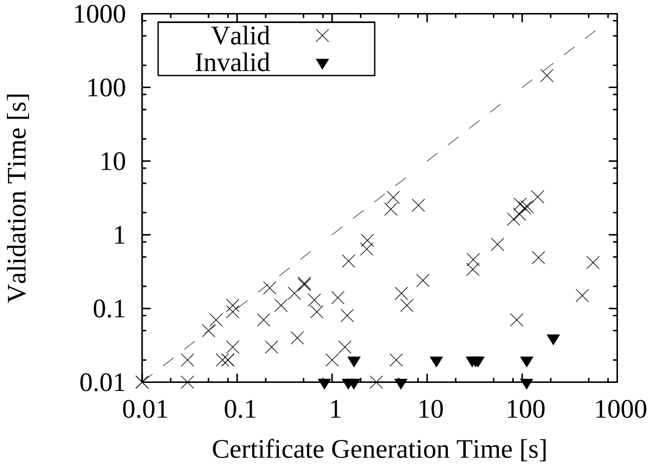
The same experiments as for the first two solvers were also run using SQUOLEM: Out of 445 instances in the original dataset, this solver finishes on 142 benchmarks; 73 instances are found to be valid, 69 invalid and certificates are produced for each of them.

Again, the time to solve an instance is compared to the time needed to generate a certificate. Naturally, in a purely Skolemization-based solver, there is a small difference in those times. Figure 7.11(a) shows that the overhead of certificate generation is usually very small (on average 3.5% for models and 4.5% for refutations, with respect to solving time).

As a second experiment, the time taken to validate a certificate is investigated. Figure 7.11(b) shows that the time taken to validate a refutation is negligibly small (on average < 0.01 seconds); certificates of validity, on the other hand, take on average 2.38 seconds to validate. The main contribution to this time is the instance ‘qshifter_7’, which takes 144 seconds to validate. Excluding this instance, the average validation time is only 0.4 seconds. The reason for this extraordinary high runtime on just one single instance is that MINISAT actually runs into a hard problem: 97% of the runtime is spent on the final model validation.



(a) Certificate Generation vs. Solver Time



(b) Certificate Generation vs. Validation Time

Figure 7.11: Performance comparison for SQUOLEM.

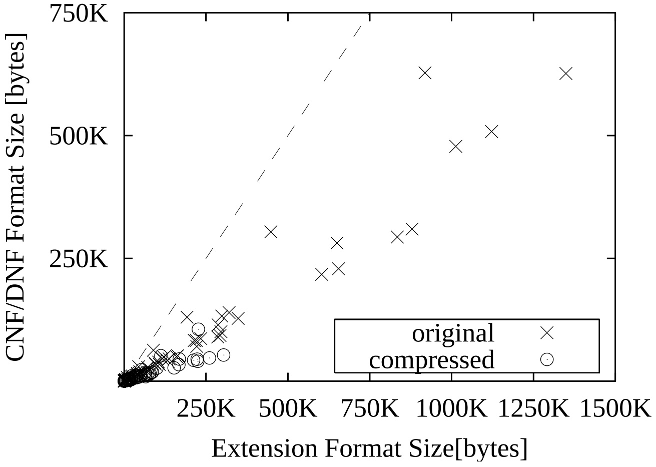
Certificate Size

An interesting property of certificates is the size of the files generated. Table 7.6 gives an overview of the relative size of the certificates generated, with respect to the size of the original formula file. The traces are represented in the original ASCII format as well as after compression with `gzip`. The data indicates that the certificates generated by EBDDRES are very large compared to those generated by QUAFFLE and SQUOLEM. Furthermore, its refutations (tracing complex BDD operations) are larger than its models. For SQUOLEM, on the other hand, the numbers suggest that models are considerably larger than refutations. The certificate format is not optimized for file size, but as the data indicates, the files compress well with `gzip`, if smaller files are required.

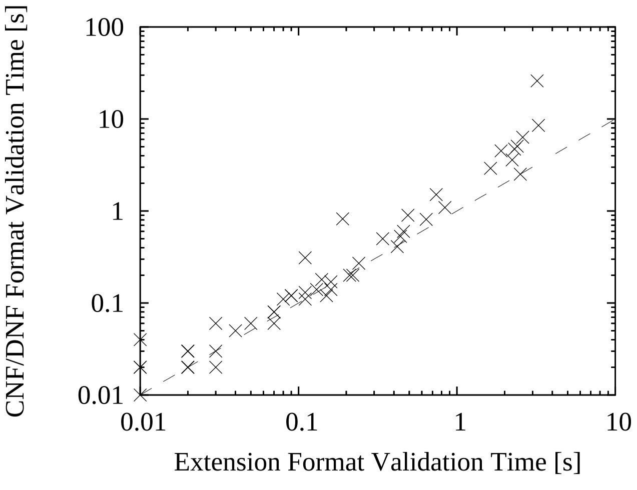
		Normal			Compressed		
Solver	Type	Best	Avg.	Worst	Best	Avg.	Worst
EBDDRES	Valid	<0.1	210.9	3304.3	<0.1	52.3	784.8
	Invalid	1.0	6857.6	145414.0	1.0	1594.3	31948.3
QUAFFLE	Valid	-	-	-	-	-	-
	Invalid	<0.1	3.4	17.0	<0.1	1.3	5.0
SQUOLEM	Valid	0.7	10.1	175.6	0.2	2.8	49.6
	Invalid	<0.1	3.3	55.0	<0.1	0.8	10.4

Table 7.6: Relative size of certificates.

For comparison, SQUOLEM can save model functions in two formats. The most straight-forward approach to saving model functions is to save them as a set of CNF or DNF definitions. Figure 7.12(a) shows a comparison of this format and the format based on extension rules. The CNF/DNF format has a slight advantage in terms of file size. However, it is harder to verify, because the function definitions have to be translated to CNF in the verifier. Figure 7.12(b) shows that the validation time advantage is clearly on the side of the extension format. The instance 'qshifter_7' is missing from this figure,



(a) Comparison of certificate size



(b) Comparison of validation time

Figure 7.12: Certificate size and validation time of the CNF/DNF certificate format compared to the extension format.

because it takes more than 600 seconds to validate in CNF/DNF format.

Finally, it is interesting to compare the solvers that were instrumented for this evaluation with fast state-of-the-art solvers. Here, QUANTOR and SKIZZO are used to demonstrate the relative performance of the solvers that generate certificates. As QUAFFLE only produces refutation traces, no data is available on valid instances for this solver. Therefore, the number of invalid instances that each solver is able to solve within 600 seconds and a 1 GB memory limit is presented here for comparison. The numbers in Table 7.7 indicate that the instrumented solvers together can solve about half as many instances as the state-of-the-art solvers are able to solve.

	EBDDRES	QUAFFLE	SQUOLEM	QUANTOR	SKIZZO
Solved Instances	80	26	69	153	175
Solved Inst. (Union)	90			178	

Table 7.7: The number of solved instances in the test set.

Alternative Models

An alternative way of giving a model for a (valid) QBF is to provide a refutation for the negation of the formula (which would then be invalid). Experiments that negate a formula by translation of the resulting DNF back to a CNF using the Tseitin-transformation show that this approach is infeasible in practice. Figure 7.13 shows a comparison of the runtime required to solve a formula in relation to the time required to solve its negation. All of the problems that are solved within 600 seconds by two state-of-the-art solvers (QUANTOR and SKIZZO) can either not be solved within the same time limit when inverted, or take considerably more time to solve.

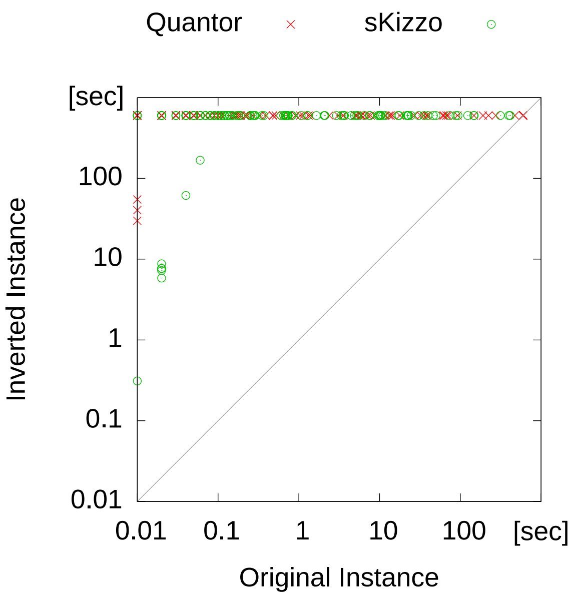
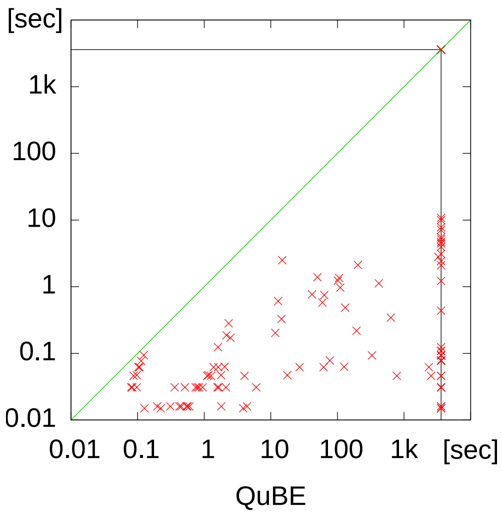


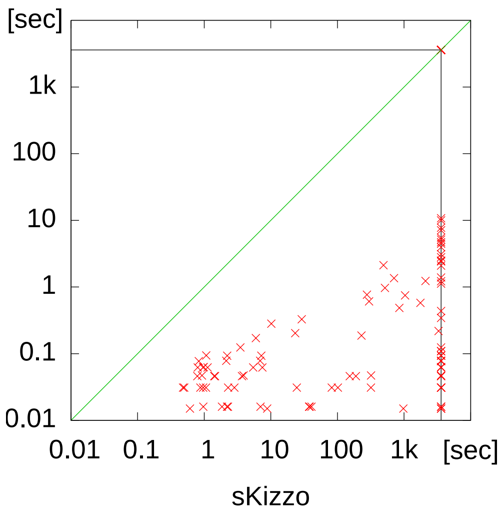
Figure 7.13: Comparison of solver performance on inverted instances.

7.3.2 Reference Implementations

EBDDRES and the instrumented version of QUAFFLE are available for download at `http://fmv.jku.at/ebddres`. SQUOLEM, the certificate validator QBV and all experimental data are available at `http://www.cprover.org/qbv`. A formal specification of the certificate format is provided in Appendix B.

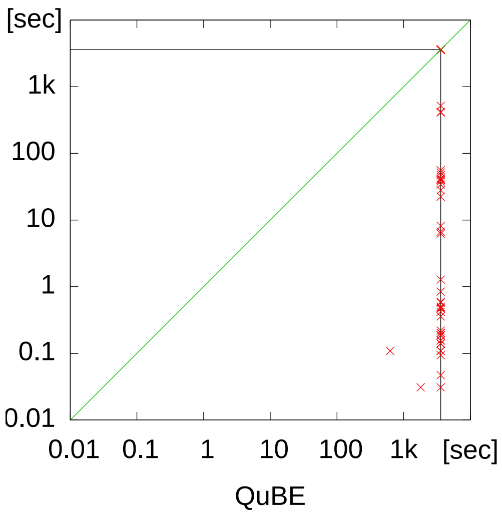


(a) QuBE vs. Z3.

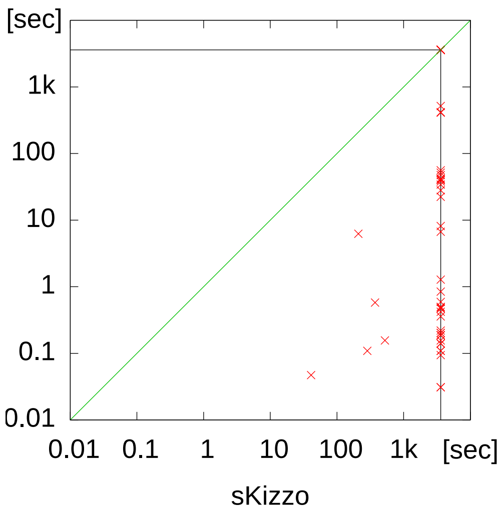


(b) sKizzo vs. Z3.

Figure 7.14: Runtime comparison on hardware fixpoint formulas.



(a) QuBE vs. Z3.



(b) sKizzo vs. Z3.

Figure 7.15: Runtime comparison on ranking function synthesis formulas.

7.4 Quantified Bit-Vector Formulas

To assess the efficacy of the QBVF solving algorithm described in Chapter 6, the algorithm was implemented using the code-base of the Z3 SMT solver [38] as a basis. This prototype first applies the simplifications described in Section 6.2.1. It then iterates model checking and model finding as described in Sections 6.3 and 6.4. The benchmarks used for the performance comparison are derived from two sources: a) hardware fixpoint checks and b) software ranking function synthesis [33]. It is not trivial to compare this QBVF solver with other systems, since most SMT solvers do not perform well on benchmarks containing bit-vectors and quantifiers. In the past, QBF solvers have been used to attack these problems. Therefore, the new prototype is compared to the state-of-the-art QBF solvers sKizzo [12] and QuBE [50] here.

Formulas in the first set exhibit the structure of fixpoint formulas described in Section 6.2. The circuits that are used as benchmarks are derived from a previous evaluation of VCEGAR [58][8] and were extracted using a customized version of the EBMC bounded Model Checker[9], which is able to produce fixpoint checks in QBVF and QBF form. In total, this benchmark set contains 131 files.

The second set of benchmarks cannot be directly encoded in QBF because they contain uninterpreted function symbols. Therefore, only ranking functions that are linear polynomials are considered, as described in Chapter 4. After applying the polynomial template, the problems are converted to QBF as described in Section 13. Thus, the problem here is to synthesise the coefficients of a polynomial. In total, this benchmark set contains 60 files.

All benchmarks were extracted in two forms: in QBVF form (using SMT-LIB syntax) and in QBF form (using the QDIMACS format) and they were executed on a Windows HPC cluster of AMD Athlon 2 GHz machines with a time limit of 1 hour (3600 seconds) and a memory limit of 2 GB.

[8]These benchmarks are available at `http://www.cprover.org/hardware/`

[9]EBMC is available at `http://www.cprover.org/ebmc/`

As indicated by the scatter plot in Figure 7.14 the new approach outperforms the QBF solvers on all hardware fixpoint instances, sometimes by up to five orders of magnitude and it solves almost all instances in the benchmark set (110 out of 131). Most of the benchmarks solved in this category (87 out of 110) are solved by the simplifications and rewriting rules only. In the remaining cases, the model refinement algorithm takes less then 10 iterations.

Figure 7.15 shows the results for the ranking function benchmark set. Again, the new algorithm outperforms the QBF solvers by up to five orders of magnitude. The number of iterations required to find a model or prove non-existence of a model in these benchmarks is again very small: almost all instances require only one or two iterations and the maximum number of iterations is 9. Even though the new algorithm exhibits similar speedups on both benchmark sets, the behaviour on the second set is quite different as none of the instances in this set is solved by simplification alone. The model finding algorithm is required on each of them.

Tables 7.8, 7.9, 7.10 and 7.11 provide all the timings (in seconds) and results of the experiments. Note that they also include the runtime of the QBF solver Quantor.

	sKizzo	QuBE	Quantor	Z3	Result
AR-fixpoint-1.qdimacs	TIME	MEM	MEM	0.077	unsat
AR-fixpoint-10.qdimacs	MEM	MEM	TIME	0.124	unsat
AR-fixpoint-2.qdimacs	TIME	MEM	MEM	0.078	unsat
AR-fixpoint-3.qdimacs	TIME	MEM	TIME	0.078	unsat
AR-fixpoint-4.qdimacs	MEM	MEM	TIME	0.078	unsat
AR-fixpoint-5.qdimacs	MEM	MEM	TIME	0.094	unsat
AR-fixpoint-6.qdimacs	MEM	MEM	TIME	0.109	unsat
AR-fixpoint-7.qdimacs	MEM	MEM	TIME	0.094	unsat
AR-fixpoint-8.qdimacs	MEM	MEM	TIME	0.109	unsat
AR-fixpoint-9.qdimacs	MEM	MEM	TIME	0.109	unsat
cache-coherence-2-fixpoint-1.qdimacs	3298.794	193.22	MEM	0.218	unsat
cache-coherence-2-fixpoint-2.qdimacs	TIME	TIME	MEM	1.217	unsat
cache-coherence-2-fixpoint-3.qdimacs	TIME	TIME	MEM	2.417	unsat
cache-coherence-2-fixpoint-4.qdimacs	TIME	TIME	MEM	3.946	unsat
cache-coherence-2-fixpoint-5.qdimacs	MEM	TIME	MEM	7.098	unsat
cache-coherence-2-fixpoint-6.qdimacs	TIME	TIME	MEM	10.748	unsat
cache-coherence-3-fixpoint-1.qdimacs	TIME	630.714	MEM	0.343	unsat
cache-coherence-3-fixpoint-2.qdimacs	TIME	TIME	MEM	2.09	unsat
cache-coherence-3-fixpoint-3.qdimacs	TIME	TIME	MEM	4.461	unsat
ethernet-fixpoint-1.qdimacs	1036.26	63.214	MEM	0.748	unsat
ethernet-fixpoint-2.qdimacs	MEM	3266.96	MEM	2.793	unsat
ethernet-fixpoint-3.qdimacs	MEM	TIME	MEM	4.696	unsat
ethernet-fixpoint-4.qdimacs	TIME	TIME	MEM	9.999	unsat
itc-b13-fixpoint-1.qdimacs	2.277	1.643	MEM	0.031	unsat
itc-b13-fixpoint-10.qdimacs	704.89	105.746	MEM	1.357	sat
itc-b13-fixpoint-2.qdimacs	5.94	2.483	MEM	0.171	unsat
itc-b13-fixpoint-3.qdimacs	23.183	11.654	MEM	0.203	sat
itc-b13-fixpoint-4.qdimacs	29.02	14.42	MEM	0.328	sat
itc-b13-fixpoint-5.qdimacs	850.897	130.657	MEM	0.484	sat
itc-b13-fixpoint-6.qdimacs	1755.936	59.454	MEM	0.577	sat
itc-b13-fixpoint-7.qdimacs	277.154	41.524	MEM	0.764	sat
itc-b13-fixpoint-8.qdimacs	515.197	109.94	MEM	0.967	sat
itc-b13-fixpoint-9.qdimacs	TIME	417.43	MEM	1.123	sat
pi-bus-fixpoint-1.qdimacs	TIME	TIME	MEM	0.437	unsat
pi-bus-fixpoint-2.qdimacs	TIME	TIME	MEM	3.089	unsat
pi-bus-fixpoint-3.qdimacs	TIME	TIME	MEM	5.132	unsat
sdlx-fixpoint-1.qdimacs	3.487	1.61	MEM	0.124	unsat
sdlx-fixpoint-10.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-2.qdimacs	10.17	2.32	MEM	0.281	unsat
sdlx-fixpoint-3.qdimacs	298.317	12.854	MEM	0.608	unsat
sdlx-fixpoint-4.qdimacs	2096.083	101.177	MEM	1.232	unsat
sdlx-fixpoint-5.qdimacs	490.48	202.53	MEM	2.121	unsat
sdlx-fixpoint-6.qdimacs	MEM	TIME	MEM	TIME	?
sdlx-fixpoint-7.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-8.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-9.qdimacs	TIME	TIME	MEM	TIME	?
small-bug1-fixpoint-1.qdimacs	0.507	0.957	0.17	0	sat
small-bug1-fixpoint-10.qdimacs	1.074	0.124	0.357	0.094	sat
small-bug1-fixpoint-2.qdimacs	0.48	0.08	0.147	0.031	sat
small-bug1-fixpoint-3.qdimacs	0.5	0.083	0.16	0.031	sat
small-bug1-fixpoint-4.qdimacs	0.787	0.087	0.163	0.046	sat
small-bug1-fixpoint-5.qdimacs	0.873	0.097	0.19	0.031	sat
small-bug1-fixpoint-6.qdimacs	0.924	0.097	0.184	0.047	sat
small-bug1-fixpoint-7.qdimacs	0.797	0.103	0.233	0.062	sat
small-bug1-fixpoint-8.qdimacs	0.96	0.107	0.21	0.062	sat
small-bug1-fixpoint-9.qdimacs	0.826	0.113	0.226	0.077	sat

Table 7.8: Timing data for hardware fixpoint checks (Part 1).

	sKizzo	QuBE	Quantor	Z3	Result
small-dyn-partition-fixpoint-1.qdimacs	0.61	0.127	0.186	0.015	unsat
small-dyn-partition-fixpoint-10.qdimacs	2.206	TIME	MEM	0.093	unsat
small-dyn-partition-fixpoint-2.qdimacs	0.963	0.513	2.42	0.031	unsat
small-dyn-partition-fixpoint-3.qdimacs	0.97	4.383	4.25	0.016	unsat
small-dyn-partition-fixpoint-4.qdimacs	1.064	6.043	33.11	0.031	unsat
small-dyn-partition-fixpoint-5.qdimacs	0.983	125.81	MEM	0.063	unsat
small-dyn-partition-fixpoint-6.qdimacs	1.41	776.993	MEM	0.046	unsat
small-dyn-partition-fixpoint-7.qdimacs	1.123	2347.22	MEM	0.062	unsat
small-dyn-partition-fixpoint-8.qdimacs	1.454	2538.42	MEM	0.046	unsat
small-dyn-partition-fixpoint-9.qdimacs	2.14	TIME	MEM	0.078	unsat
small-equiv-fixpoint-1.qdimacs	MEM	TIME	MEM	0.015	sat
small-equiv-fixpoint-10.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-2.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-3.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-4.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-5.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-6.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-7.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-8.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-9.qdimacs	MEM	TIME	MEM	TIME	?
small-pipeline-fixpoint-1.qdimacs	TIME	TIME	MEM	0.016	unsat
small-pipeline-fixpoint-10.qdimacs	MEM	TIME	MEM	TIME	?
small-pipeline-fixpoint-2.qdimacs	TIME	TIME	MEM	0.031	unsat
small-pipeline-fixpoint-3.qdimacs	TIME	TIME	MEM	0.093	unsat
small-pipeline-fixpoint-4.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-5.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-6.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-7.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-8.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-9.qdimacs	MEM	TIME	MEM	TIME	?
small-seq-fixpoint-1.qdimacs	976.613	3.84	MEM	0.015	unsat
small-seq-fixpoint-10.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-2.qdimacs	MEM	TIME	MEM	0.015	unsat
small-seq-fixpoint-3.qdimacs	MEM	TIME	MEM	0.016	unsat
small-seq-fixpoint-4.qdimacs	TIME	TIME	MEM	0.015	unsat
small-seq-fixpoint-5.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-6.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-7.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-8.qdimacs	TIME	TIME	MEM	0.046	unsat
small-seq-fixpoint-9.qdimacs	TIME	TIME	MEM	0.046	unsat
small-swap1-fixpoint-1.qdimacs	8.813	0.223	MEM	0.015	unsat
small-swap1-fixpoint-10.qdimacs	TIME	2.02	MEM	0.063	sat
small-swap1-fixpoint-2.qdimacs	24.506	0.36	MEM	0.031	sat
small-swap1-fixpoint-3.qdimacs	37.483	0.427	MEM	0.016	sat
small-swap1-fixpoint-4.qdimacs	TIME	0.6	MEM	0.016	sat
small-swap1-fixpoint-5.qdimacs	TIME	0.79	MEM	0.031	sat
small-swap1-fixpoint-6.qdimacs	MEM	0.947	MEM	0.031	sat
small-swap1-fixpoint-7.qdimacs	TIME	1.157	MEM	0.047	sat
small-swap1-fixpoint-8.qdimacs	TIME	1.383	MEM	0.062	sat
small-swap1-fixpoint-9.qdimacs	TIME	1.626	MEM	0.062	sat
small-swap2-fixpoint-1.qdimacs	0.55	0.163	MEM	0	unsat
small-swap2-fixpoint-10.qdimacs	319.853	1.787	MEM	0.047	sat
small-swap2-fixpoint-2.qdimacs	7.007	0.31	MEM	0.016	unsat
small-swap2-fixpoint-3.qdimacs	38.127	0.45	MEM	0.016	sat
small-swap2-fixpoint-4.qdimacs	40.767	0.543	MEM	0.016	sat
small-swap2-fixpoint-5.qdimacs	101.52	0.746	MEM	0.031	sat
small-swap2-fixpoint-6.qdimacs	81.994	0.836	MEM	0.031	sat
small-swap2-fixpoint-7.qdimacs	152.68	1.11	MEM	0.046	sat
small-swap2-fixpoint-8.qdimacs	188.513	1.267	MEM	0.046	sat
small-swap2-fixpoint-9.qdimacs	318.016	1.576	MEM	0.031	sat

Table 7.9: Timing data for hardware fixpoint checks (Part 2).

	sKizzo	QuBE	Quantor	Z3	Result
small-synabs-fixpoint-1.qdimacs	2.2	0.197	0.523	0.016	unsat
small-synabs-fixpoint-10.qdimacs	7.203	329.67	MEM	0.093	unsat
small-synabs-fixpoint-2.qdimacs	1.843	0.563	MEM	0.016	unsat
small-synabs-fixpoint-3.qdimacs	2.273	1.806	MEM	0.016	unsat
small-synabs-fixpoint-4.qdimacs	2.83	2.117	MEM	0.031	unsat
small-synabs-fixpoint-5.qdimacs	3.693	4.03	MEM	0.046	unsat
small-synabs-fixpoint-6.qdimacs	3.887	17.686	MEM	0.047	unsat
small-synabs-fixpoint-7.qdimacs	5.437	26.947	MEM	0.062	unsat
small-synabs-fixpoint-8.qdimacs	7.447	61.907	MEM	0.062	unsat
small-synabs-fixpoint-9.qdimacs	6.907	76.893	MEM	0.078	unsat
usb-phy-fixpoint-1.qdimacs	229.333	2.163	MEM	0.187	unsat
usb-phy-fixpoint-2.qdimacs	TIME	50.123	MEM	1.388	unsat
usb-phy-fixpoint-3.qdimacs	TIME	14.86	MEM	2.496	unsat
usb-phy-fixpoint-4.qdimacs	TIME	TIME	MEM	5.491	unsat
usb-phy-fixpoint-5.qdimacs	TIME	TIME	MEM	7.753	unsat

Table 7.10: Timing data for hardware fixpoint checks (Part 3).

	sKizzo	QuBE	Quantor	Z3	Result
1394diag_ioctl.c.qdimacs	TIME	TIME	MEM	TIME	?
1394diag_isochapi.c.qdimacs	MEM	TIME	MEM	40.591	sat
audio_ac97_common.cpp.qdimacs	MEM	TIME	MEM	0.468	sat
audio_ac97_rtstream.cpp.qdimacs	MEM	TIME	MEM	0.202	sat
audio_ac97_wavepcistream.cpp.qdimacs	TIME	TIME	MEM	416.757	unsat
audio_ac97_wavepcistream2.cpp.qdimacs	MEM	TIME	MEM	0.483	unsat
audio_ac97_wavepcistream3.cpp.qdimacs	MEM	TIME	MEM	0.219	unsat
audio_ddksynth_csynth.cpp.qdimacs	MEM	TIME	MEM	0.358	unsat
audio_ddksynth_csynth2.cpp.qdimacs	MEM	TIME	MEM	0.094	sat
audio_ddksynth_voice.cpp.qdimacs	TIME	TIME	MEM	28.08	unsat
audio_dmusuart_mpu.cpp.qdimacs	TIME	TIME	MEM	34.179	sat
audio_fmsynth_miniport.cpp.qdimacs	MEM	TIME	MEM	0.156	sat
audio_fmsynth_miniport2.cpp.qdimacs	MEM	626.356	MEM	0.109	sat
audio_gfxswap.xp_filter.cpp.qdimacs	MEM	TIME	MEM	0.592	unsat
audio_sysfx_swap.cpp.qdimacs	MEM	TIME	MEM	TIME	?
AVStream_hwsim.cpp.qdimacs	TIME	TIME	MEM	TIME	?
AVStream_image.cpp.qdimacs	MEM	TIME	MEM	22.401	sat
filesys_cdfs_allocsup.c.qdimacs	MEM	TIME	TIME	TIME	?
filesys_cdfs_cddata.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_cdfs_namesup.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_cdfs_namesup2.c.qdimacs	MEM	TIME	MEM	0.14	sat
filesys_fastfat_allocsup.c.qdimacs	MEM	TIME	MEM	0.187	sat
filesys_fastfat_cachesup.c.qdimacs	MEM	TIME	MEM	0.202	sat
filesys_fastfat_easup.c.qdimacs	MEM	TIME	MEM	6.676	sat
filesys_fastfat_write.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_filter_namelookup.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_smbmrx_cvsndrcv.c.qdimacs	523.737	TIME	MEM	0.156	unsat
filesys_smbmrx_midatlas.c.qdimacs	40.877	TIME	MEM	0.047	unsat
filesys_smbmrx_smbxchng.c.qdimacs	MEM	TIME	MEM	55.816	unsat
general_pcidrv_sys_hw_eeprom.c.qdimacs	MEM	TIME	MEM	0.843	unsat
general_pcidrv_sys_hw_eeprom2.c.qdimacs	MEM	TIME	MEM	0.499	sat
general_toaster_exe_notify_notify.c.qdimacs	MEM	TIME	MEM	TIME	?
hid_firefly_app_firefly.cpp.qdimacs	TIME	TIME	MEM	TIME	?
hid_hclient_ecdisp.c.qdimacs	MEM	TIME	MEM	40.108	sat
input_mouser_cseries.c.qdimacs	MEM	TIME	MEM	0.421	sat
input_mouser_detect.c.qdimacs	MEM	1796.263	MEM	0.031	sat
input_pnpi8042_moudep.c.qdimacs	MEM	TIME	MEM	51.377	sat
ir_smscir_io.c.qdimacs	MEM	TIME	MEM	TIME	?
kernel_agplib_init.c.qdimacs	MEM	TIME	MEM	0.109	sat
kernel_agplib_intrface.c.qdimacs	MEM	TIME	MEM	0.187	sat
kernel_uagp35_gart.c.qdimacs	MEM	TIME	MEM	40.107	sat
kmdf_AMCC5933_sys_S5933DK1.c.qdimacs	MEM	TIME	MEM	0.141	sat
kmdf_osrusbfx2_exe_dump.c.qdimacs	MEM	TIME	MEM	47.411	unsat
kmdf_osrusbfx2_exe_testapp.c.qdimacs	MEM	TIME	MEM	TIME	?
kmdf_pcidrv_sys_hw_nic_init.c.qdimacs	MEM	TIME	MEM	38.016	sat
kmdf_pcidrv_sys_hw_physet.c.qdimacs	MEM	TIME	MEM	0.031	sat
kmdf_usbsamp_sys_queue.c.qdimacs	MEM	TIME	MEM	TIME	?
mmedia_gsm610_gsm610.c.qdimacs	MEM	TIME	MEM	0.187	sat
mmedia_gsm610_gsm6102.c.qdimacs	284.807	TIME	MEM	0.109	unsat
mmedia_gsm610_gsm6103.c.qdimacs	371.61	TIME	MEM	0.577	unsat
mmedia_imaadpcm_imaadpcm.c.qdimacs	MEM	TIME	MEM	TIME	?
network_irda_miniport_nscirda_comm.c.qdimacs	MEM	TIME	MEM	408.901	unsat
network_irda_miniport_nscirda_settings.c.qdimacs	MEM	TIME	MEM	515.143	unsat
network_ndis_coisdn_TpiParam.c.qdimacs	MEM	TIME	MEM	8.158	sat
network_ndis_e100bex_5x_kd_mp_dbg.c.qdimacs	MEM	TIME	MEM	TIME	?
network_ndis_rtlnwifi_extsta_st_aplst.c.qdimacs	MEM	TIME	MEM	42.028	sat
network_ndis_rtlnwifi_extsta_st_misc.c.qdimacs	TIME	TIME	MEM	TIME	?
network_ndis_rtlnwifi_hw_hw_ccmp.c.qdimacs	MEM	TIME	MEM	1.279	sat
network_trans_sys_notify.c.qdimacs	209.523	TIME	MEM	6.224	unsat

Table 7.11: Timing data for ranking function synthesis checks.

Chapter 8

Conclusion

This dissertation presents a new method for termination checking called Compositional Termination Analysis. It eliminates the costly safety check in the Terminator algorithm [35] and thereby outperforms this algorithm in practice. Both algorithms require underlying methods for ranking relation synthesis and two methods for the special case of (finite-state) Bit-vector programs are presented. When combined, these techniques present a very efficient technique for termination checking of embedded software or operation system components like device drivers.

A substantial experimental evaluation of Compositional Termination Analysis indicates an average speedup of 52 over the Terminator algorithm when using the same ranking relation synthesis engine. This result makes Compositional Termination Analysis the fastest available method for termination analysis and enables the execution of termination checks as a routine step in the software design process.

While ranking relation synthesis methods are available for a number of domains, efficient procedures for programs over Bit-vectors (or machine integers) have not until now been available. This dissertation presents two new algorithms which fill this gap: a complete method based on a reduction to quantifier-free Presburger arithmetic (and Integer Linear Programming) and a template-matching method for finding ranking functions of specified classes. Through experimentation with examples drawn from Windows device drivers their efficiency and applicability to systems-level code is demonstrated. In their current state, the bottleneck with these methods is the reachability analysis engine. Future research therefore should consider optimizations of the reachability checker or new procedures specific to reachability analysis in the context of termination proving.

Many sub-problems of termination checking or the underlying ranking relation synthesis algorithms may be solved using decision procedures for the validity problem of quantified Boolean formulae (QBF). These decision procedures however, are not usually able to produce certificates for their answers. This dissertation demonstrates that it is possible to define a proof format for QBF that is applicable to a wide range of different QBF decision procedures. Nevertheless, important common features of other QBF decision procedures cannot be traced efficiently in this format. Clearly, there is considerable scope for future work that extends or replaces this certificate format. Modern QBF decision procedures make decision based on rules which do not have an immediate equivalent in the certificate format and it is conjectured that these require stronger proof rules than the resolution calculus combined with the quantified extension rule presented in this dissertation.

To circumvent the performance and certification problems of QBF decision procedures, an alternative is presented in the form of a different logic that is a superset of QBF. Quantified bit-vector logic (QBV) is ideally suited as an interface between verification or synthesis tools and underlying decision procedures. Different fragments of this logic are required in virtually every verification or synthesis technique known, making QBV one of the most practically relevant logics. This dissertation presents a new approach to solving quantified bit-vector formulas based on a set of simplifications and rewrite rules, as well as a new model finding algorithm based on an iterative refinement scheme. Extensive experimental evaluation indicates that this decision procedure is up to five orders of magnitude faster when compared to modern QBF decision procedures, while certification of the decision procedure becomes a trivial matter.

Combined, the methods presented in this theses present an integrated solution to bit-precise termination analysis of low-level and embedded software. Through extensive experimental work this solution is proven to be a viable alternative to testing in the design process which identifies termination bugs efficiently and effectively. Through the increase in performance over existing methods, checking software for termination as a routine step before it is released becomes a practical possibility. Perhaps this demonstration of utility could contribute to such checking becoming common practice.

Appendix A

Typing Rules for Bit-Vector Programs

We write $t : n$ to denote that the expression t is correctly typed and denotes a bit-vector of length n. Given a statement or program β, we write $\beta : \bot$ to express that β is correctly typed. In the following rules, $x \in \mathcal{X}$ ranges over variables, $n \in \mathbb{N}^+$ over positive natural numbers, s, t over expressions, β, γ over statements:

$$\frac{k \in \mathbb{N}^+}{k_n : n} \quad \frac{}{*_n : n} \quad \frac{t : n}{\neg t : n} \quad \frac{s : n \quad t : n}{s \circ t : n}\ \circ \in \{+, \times, \div, \ \&\ , \mid\}$$

$$\frac{\alpha(x) = n}{x : n} \qquad \frac{s : n \quad t : k}{s \circ t : n}\ \circ \in \{\ll, \gg\} \qquad \frac{\alpha(x) = n \quad t : n}{x := t : \bot}$$

$$\frac{t : k}{\mathsf{cast}_n(t) : n} \qquad \frac{s : n \quad t : n}{s \circ t : 1}\ \circ \in \{=, \leq\} \qquad \frac{t : 1}{\mathsf{assume}\ (t) : \bot}$$

$$\frac{}{\mathsf{skip} : \bot} \quad \frac{\beta : \bot \quad \gamma : \bot}{\beta; \gamma : \bot} \quad \frac{\beta : \bot \quad \gamma : \bot}{\beta \,\square\, \gamma : \bot} \quad \frac{\beta : \bot \quad \gamma : \bot}{\beta\ \mathsf{repeat}\ \{\,\gamma\,\} : \bot}$$

Appendix B

The QB Certificate File Format

The following sections define a proposed file format for certificates of QBF instances.

B.1 Definitions

A *variable* in any QBF problem is represented by its variable index; an integer in the range $[1, 2^{31}]$.

A *literal* is a variable in either negated or non-negated form. A literal is represented by an integer in the range $[-2^{31}, 2^{31}]\backslash\{0\}$, where negative numbers represent negated variables.

A *clause* is a disjunction of literals and is represented by a set of literals. No order of literals is assumed. Every clause implicitly has a clause index; all clauses of a QBF problem are numbered ascending in order of appearence in the according problem file. Clause indices are integers from the range $[1, 2^{32}]$.

The QBF problem is assumed to be given in a file that uses the QDIMACS format.

B.2 Basic Rules

```
        NL → the newline character ('\n').
     DIGIT → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9).
    LETTER → (a | ... | z | A | ... | Z).
 EXTRACHAR → ( . | _ | $ | ... ).
  FILENAME → (LETTER | DIGIT | EXTRACHAR)
             { LETTER | DIGIT | EXTRACHAR }.
    NUMBER → DIGIT { DIGIT }.
    CINDEX → NUMBER.
    VINDEX → NUMBER.
   LITERAL → [-] VINDEX
    CCOUNT → NUMBER.
```

Note: A NUMBER is always assumed to be an integer in the range $[1, 2^{32}]$, except for the VINDEX which refers to a variable index and thus is in the range $[1, 2^{31}]$.

B.3 Header

Every QB Certificate file is required to begin with an appropriate header line that contains the `QBCertificate` keyword.

I.e.,

```
HEADER → QBCertificate NL.
```

B.4 Extensions

Based on the Extension Theorem (Chapter 5 and [62]), extension rules are supported. Instead of allowing the introduction of arbitrary functions (which would be too hard to verify), only two important types are supported. An extension line has the format

EXTENSION → ITE | AND.

where the first VINDEX defines a fresh variable that will be quantified existentially and in the scope of the innermost variable that appears in the extension.

If-Then-Else. The If-Then-Else extension rule allows the introduction of a new variable that is defined to be the If-Then-Else of three existing variables. The If-Then-Else extension thus requires exactly three parameters:

ITE → `E` VINDEX `I` LITERAL LITERAL LITERAL NL.

This rule will introduce four new clauses into the original formula; for *w = if x then a else b* the clauses $(\neg w \vee \neg x \vee a)$, $(\neg w \vee x \vee b)$, $(w \vee \neg x \vee \neg a)$ and $(w \vee x \vee \neg b)$ are introduced.

AND. A new variable may be defined to be the logical AND of existing variables:

AND → `E` VINDEX `A` { LITERAL } `0` NL.

The number of existing variables that are used in this definition is not limited, but the extension line must be terminated by a 0.

This rule will introduce $n + 1$ clauses into the original formula: $(\neg f \vee v_1)$ to $(\neg f \vee v_n)$ and $(f \vee \neg v_1 \vee \cdots \vee \neg v_n)$, where f is the new variable that depends on the existing variables v_i.

B.5 Resolution

Whenever a clause should be (q-)resolved against another, this must be listed in the certificate. The resolution line format is the same as used in traces for SAT–Solvers.

I.e.,

```
RESOLUTION → CINDEX
             ( * | ({ LITERAL } 0) )
             { CINDEX } 0 NL.
```

The first CINDEX in this rule defines the new clause index that the resolvent should be assigned. What follows is the resolvent (a sequence of LITERALs, terminated by a 0) and the antecedents that the desired resolvent can be calculated from (a sequence of CINDEXs, terminated by a 0). The actual order of the resolutions will be determined by the verifier via constraint propagation.

Note that a q-resolution step includes forall-reduction.

B.6 Conclusions

The last line in any QB Certificate is the conclusion line that states whether the original problem is to be proven valid or invalid. In case of an unsatisfiable problem the index of the clause which should be empty, must be given.

I.e.,

```
CONCLUSION → CONCLUDE (VALID {VINDEX LITERAL} |
                       INVALID CINDEX ).
```

For valid instances, the model is to be supplied in the conclusion line. New variables have to be defined by extensions first. The conclusion line will then contain a set of equivalences $e = n$, where e is an original, existentially quantified variable and n is an extension variable. These equivalences are to be supplied as a set of VINDEX/LITERAL pairs in the conclusion line.

B.7 The QB Certificate

Finally, a complete QB Certificate is defined as

```
CERTIFICATE → HEADER
             {RESOLUTION |
              EXTENSION }
              CONCLUSION.
```

Appendix C

The GOTO-CC Model Extractor

One of the problems of model checking in an industrial environment is that most software sources come in the form of multiple source files, which are compiled separately and then linked together to form the binaries. This is especially the case for the vast number of open source projects written in C and C++, which are an enormous source of benchmarks for model checkers, while at the same time they could benefit from researchers applying their newest technologies to them and then reporting any bugs they find.

Most model checking tools are also designed to handle one input language only, which restricts them to being used in a specialized field of model checking. While new techniques are invented, implemented and tested for the one language that the model checker supports, they often are not implemented in other model checkers that support different languages. Because of this, certain model checking techniques are only used in a very small environment and only on a special type of program or model, even though they might be useful on other programs (and languages) as well.

To overcome these issues, we have designed GOTO-CC, a multi-frontend model extractor, which generates GOTO-programs from source code. The result is a *model file* in binary form. Alternatively, GOTO-CC extracts a model file in XML format, which allows for easier debugging. We parse the source files (currently C and C++ are supported), compile them into separate object files, which we link together to form a binary file, much like a compiler would do. In turn, this file can be loaded into one of our model checkers, e.g., CBMC or SATABS [27].

Introducing a separate compilation step reduces the demands on the model checker, as it does not have to support various complicated languages. In scenarios that require an input file to be loaded frequently, e.g., when the source base that is being analyzed is under active development, the load times are reduced, as unmodified modules do not have to be recompiled. GOTO-CC fully supports library linking, such that it is possible to extract and use library models.

Although produced using similar processes as binary files, GOTO-CC model files are not machine code binaries, but representations of the original source code. These models include the complete symbol table that was generated during parsing and all functions, translated into GOTO-program format. Thus, the only control-flow structure used in this format is the GOTO instruction, very much like in machine code. While this seems to be a restriction at first glance, it simplifies the model checking algorithms that operate on the model. One could argue that information about the structure of the code is lost, but all this information can be preserved in the form of annotations within the model.

Another helpful feature of GOTO-CC is the generation of flowgraphs for functions. By employing the `-dot` command-line option, goto-cc produces a Graphviz compatible file, that contains a graphical description of every function in a module. Figure C.1 shows an example of some C language source code and the generated control flow graph.

Related Work: GMC^2 [52] is a software model checker that is based on the internal representation format of the new version of GCC (formerly called Tree-SSA branch). This version of GCC features two new intermediate languages called GENERIC and GIMPLE, which can be used by other programs to analyze any program in one of the six GCC input languages. GENERIC is meant to be used for abstract syntax tree analysis and optimization, while GIMPLE can be used for control flow analysis and optimization.

By basing GMC^2 on these intermediate languages, the model checker is simplified and the authors can use six different input languages at once. However, the model checker is bound to use the information that it gets from GENERIC and GIMPLE, which sadly is not always sufficient for every kind of analysis. Also, the static analyzers MOPS [23] and COVERITY [43] are both tightly integrated with the GCC build process.

A popular tool for the C language is the CIL (C Intermediate Language) [76] parser library. It can be used as a preprocessor and parser for C programs and supports most of the GCC and Microsoft extensions. Where support for C as an input language is required, CIL can be included and its parse trees used directly, or CIL may be used to transform C code into a small fragment of C, which is considerably easier to handle than full C. CIL, however, can only be used for the C language and does not support C++. Examples of applications that use CIL include the F-Soft Software Verification Platform [57], and BLAST [55].

C.1 Separating Model Extraction

Compiling and linking the modules of a software project can become a very complex task. The most commonly used build tool, which automates this process, is the *make* utility, which is the most prevalent build tool on Unix and Linux systems. Even Microsoft's *nmake* utility, which comes with their Visual Studio Suite and is used to compile large software projects for the Windows

```
extern int getch( void );
extern int printf( const char *, ...);

int main( void ) {
  char a;
  a = getch();
  while (a!='\n') {
    switch (a) {
      case 'a':
      case 'b':
        printf("a␣or␣b");
      break;
      case 'c':
        printf("c␣and␣");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```

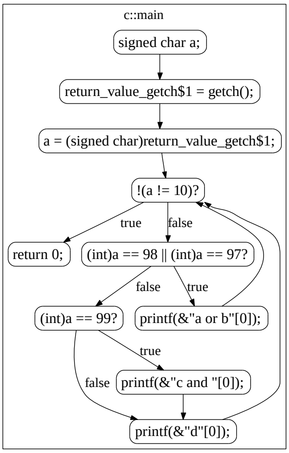


Figure C.1: Fragment of a C program and its corresponding, automatically generated, control flow graph.

platform, uses a very similar configuration file layout.

When looking at software projects implemented in languages like C or C++, the most common case is that of a set of .c or .cpp files, accompanied by a file called *Makefile*, which describes how to generate a binary from the source files. Often the makefile doesn't come ready-made with the source code, but is generated by a script called *configure*, which enables the user to set additional options and then sets the compilation options according to the systems configuration.

In practice using a model checker within the build process is problematic; usually it is necessary to replicate the build tools functionality to collect and configure source code in the model checker. Using a model extractor like GOTO-CC simplifies this step by first producing a model of the software project.

The model is a direct derivation of the original source code and could theoretically even be used to compile an executable from it. Nonetheless, the model is a greatly simplified version of the original source code, meant to simplify the model checking algorithms that work on it.

In addition to translating the input programs into programs that have a simplified control flow, i.e., they only use the (conditional) GOTO instruction, GOTO-CC also simplifies expressions and propagates constants.

C.2 Conclusion and Future Work

By compiling a software project that comes as a set of source code files into a model file prior to feeding it to a model checker, we gain flexibility and simplify the application of model checking algorithms to industrial software projects. We also provide some evidence of the practical usability of our tool[1]:

Name	**LOC**	**Description**
gnupg-1.4.4	142,450	The GNU Privacy Guard.
sendmail-8.13.8	129,864	A popular E-Mail server.
inn-2.4.3	125,891	The internet-news daemon.
wu-ftpd-2.6.2	35,311	A widely used FTP server.
limmat-1.3	8,477	A SAT solver by A. Biere.

In the future, we plan to work on compatibility issues regarding GCC, Microsoft's C-Compiler, and the respective system libraries until GOTO-CC can seamlessly replace them when switching from debug mode to "verification mode". We plan to further improve the performance and to build a reference set of libraries and applications that model checkers can be tested against.

[1] Lines of code were measured by running `find . -name "*.[ch]" | xargs wc -l`

C.3 Availability

GOTO-CC is available for download at `http://www.cprover.org/goto-cc/`. We currently distribute binaries for Linux and Windows, as well as a library of ANSI-C header files to replace the original system headers in case of compatibility issues. Our website also features a constantly expanding list of example applications, model files and instructions on how to extract model files.

List of Tables

List of Figures

Bibliography

[1] E. Ashcroft and Z. Manna. The translation of ’go to’ programs to ’while’ programs. In *Classics in software engineering*, pages 49–61. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[2] Abdelwaheb Ayari and David A. Basin. QUBOS: Deciding quantified Boolean logic using propositional satisfiability solvers. In *Proc. of FM-CAD*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[4] Domagoj Babic, Alan J. Hu, Zvonimir Rakamaric, and Byron Cook. Proving termination by divergence. In *Proc. of SEFM*, pages 93–102. IEEE, 2007.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. of EuroSys*, pages 73–85. ACM, 2006.

[6] Thomas Ball, Orna Kupferman, and Mooly Sagiv. Leaping loops in the presence of abstraction. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 491–503. Springer, 2007.

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

[8] Clark Barrett and Cesare Tinelli. CVC3. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[9] Marco Benedetti. Evaluating QBFs via symbolic skolemization. In *Proc. of LPAR*, volume 3452 of *LNCS*, pages 285–300. Springer, 2005.

[10] Marco Benedetti. Experimenting with QBF-based formal verification. In *Proc. of the 3rd Intl. Workshop on Constraints in Formal Verification (CFV)*, 2005.

[11] Marco Benedetti. Extracting certificates from quantified Boolean formulas. In *Proc. of IJCAI*, pages 47–53, 2005.

[12] Marco Benedetti. sKizzo: A suite to evaluate and certify QBFs. In *Proc. of CADE*, volume 3632 of *LNCS*, pages 369–376. Springer, 2005.

[13] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. *SIGPLAN Not.*, 42(1):211–224, 2007.

[14] Armin Biere. Resolve and expand. In *Proc. of SAT'04 (Revised Selected Papers)*, volume 3542 of *LNCS*. Springer, 2005.

[15] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *Proc. of FMICS*, volume 66 of *ENTCS*, pages 160–177. Elsevier, 2002.

[16] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[17] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. of CAV*, volume 3576 of *LNCS*, pages 491–504. Springer, 2005.

[18] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *Proc. of CONCUR*, volume 3653 of *LNCS*, pages 488–502. Springer, 2005.

[19] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. of TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.

[20] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 299–303. Springer, 2008.

[21] Hans Kleine Büning, K. Subramani, and Xishun Zhao. On boolean models for quantified boolean horn formulas. In *Proc. of SAT*, volume 2919 of *LNCS*, pages 93–104. Springer, 2003.

[22] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proc. of AAAI/IAAI*, pages 262–267. AAAI, 1998.

[23] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Conference on Computer and Communication Security (CCS)*, pages 235–244. ACM Press, 2002.

[24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Y. Lu, and Helmuth Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[25] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[26] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[27] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. of TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[28] Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Alg. and Log. Programming*, pages 31–45. Springer, 1997.

[29] Michael Colón. Schema-guided synthesis of imperative programs by constraint solving. In *Proc. of Intl. Symp. on Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 166–181. Springer, 2005.

[30] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[31] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *Proc. of CAV*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

[32] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 328–340. Springer, 2008.

[33] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proc. of TACAS*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.

[34] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Proc. of SAS*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

[35] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. of PLDI*, pages 415–426. ACM, 2006.

[36] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. of PLDI*, pages 415–426. ACM, 2006.

[37] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Proc. of CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[38] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[39] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.

[40] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with QBF. In *Proc. of SAT*, volume 3569 of *LNCS*, pages 408–414. Springer, 2005.

[41] Uwe Egly, Martina Seidl, and Stefan Woltran. A solver for QBFs in negation normal form. *Constraints*, 14(1):38–79, 2009.

[42] Emmanuelle Encrenaz and Alain Finkel. Automatic verification of counter systems with ranking functions. In *Proc. of the 11th Intl. Workshop on Verification of Infinite-State Systems (INFINITY)*, volume 239 of *ENTCS*, pages 85–103. Elsevier, 2009.

[43] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of OSDI*, pages 1–16. USENIX, 2000.

[44] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[46] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[47] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In *Rewriting Tech. and Appl.*, pages 210–220. Springer, 2004.

[48] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE: A system for deciding quantified Boolean formulas satisfiability. In *Proc. of IJCAR*, volume 2083 of *LNCS*, pages 364–369. Springer, 2001.

[49] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QBF reasoning on real–world instances. In *Proc. of SAT*, volume 3542 of *LNCS*, pages 105–121. Springer, 2004.

[50] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE++: An efficient QBF solver. In *Proc. of FMCAD*, volume 3312 of *LNCS*, pages 201–213. Springer, 2004.

[51] Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. Beyond CNF: A circuit-based QBF solver. In *Proc. of SAT*, volume 5584 of *LNCS*, pages 412–426. Springer, 2009.

[52] Radu Grosu and Scott Smolka. Monte carlo model checking. In *Proc. of TACAS*, volume 3440 of *LNCS*, pages 271–286. Springer, 2005.

[53] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Proc. of VMCAI*, volume 5403 of *LNCS*, pages 120–135. Springer, 2009.

[54] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[55] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proc. of the SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[56] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proc. of PLDI*, pages 35–46. ACM, 1988.

[57] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-SOFT: Software verification platform. In *Proc. of CAV*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.

[58] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. Word-level predicate-abstraction and refinement techniques for verifying RTL verilog. *Trans. on CAD of Int. Circuits and Systems*, 27(2):366–379, 2008.

[59] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proc. of ICSE*, pages 215–224. ACM, 2010.

[60] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Proc. of FMCAD*, pages 117–124. IEEE, 2006.

[61] Toni Jussila and Armin Biere. Compressing BMC encodings with QBF. *ENTCS*, 174(3):45–56, 2007.

[62] Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kroening, and Christoph M. Wintersteiger. A first step towards a unified proof checker for QBF. In *SAT*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.

[63] Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Proc. of SAT*, volume 4121 of *LNCS*, pages 54–60. Springer, 2006.

[64] Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for quantified Boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.

[65] Hans Kleine Büning and Xishun Zhao. On models for quantified Boolean formulas. In *Logic versus Approximation*, volume 3075 of *LNCS*, pages 18–32. Springer, 2004.

[66] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebra. In *Proc. Conf. on Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[67] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proc. of CAV*, volume 6174 of *LNCS*, pages 89–103. Springer, 2010.

[68] Richard E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

[69] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proc. of POPL*, pages 81–92. ACM, 2001.

[70] Reinhold Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *Proc. of TABLEAUX*, volume 2381 of *LNCS*, pages 160–175, 2002.

[71] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In *Proc. of CAV*, volume 1254 of *LNCS*, pages 444–447, 1997.

[72] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. BAT: The bit-level analysis tool. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 303–306. Springer, 2007.

[73] Dmitry Mirimanoff. Les antimonies de Russell et de Burali-Forti et le problème fondamental de la théorie des ensembles. *L'Enseignement Mathématique*, 19:37–52, 1927.

[74] F.L. Morris and Clifford B. Jones. An Early Program Proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, April 1984.

[75] Massimo Narizzano, Armando Tacchella, and Luca Pulina. Report of the third QBF solvers evaluation. *JSAT*, 2:145–164, 2006.

[76] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of CC*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[77] Charles Otwell, Anja Remshagen, and Klaus Truemper. An effective QBF solver for planning problems. In *Proc. of MSV/AMCS*, pages 311–316. CSREA Press, 2004.

[78] Guoqiang Pan and Moshe Y. Vardi. Symbolic decision procedures for QBF. In *Proc. of CP*, volume 3258 of *LNCS*, pages 453–467. Springer, 2004.

[79] David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Appl. Math.*, 130(2):291–328, 2003.

[80] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. of POPL*, pages 179–190. ACM, 1989.

[81] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. of VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[82] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proc. of LICS*, pages 32–41. IEEE, 2004.

[83] Andreas Podelski and Andrey Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Proc. of Symp. on Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *LNCS*, pages 245–259. Springer, 2007.

[84] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warsaw 1929*, pages 92–101, 1930.

[85] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[86] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proc. of LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[87] Horst Samulowitz and Fahiem Bacchus. Using SAT in QBF. In *Proc. of CP*, volume 3709 of *LNCS*, pages 578–592. Springer, 2005.

[88] Horst Samulowitz and Fahiem Bacchus. Binary clause reasoning in QBF. In *Proc. of SAT*, volume 4121 of *LNCS*, pages 353–367. Springer, 2006.

[89] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[90] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In *Proc. of the 1st Intl. Computer Science Symp. in Russia (CSR 2006)*, volume 3967 of *LNCS*, pages 600–611. Springer, 2006.

[91] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proc. of PLDI*, pages 223–234. ACM, 2009.

[92] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proc. of POPL*, pages 313–326. ACM, 2010.

[93] Stefan Staber and Roderick Bloem. Fault localization and correction with QBF. In *SAT*, volume 4501 of *LNCS*, pages 355–368. Springer, 2007.

[94] Larry J. Stockmeyer. The polynomial–time hierarchy. *TCS*, 3:1–22, 1976.

[95] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time (preliminary report). In *Proc. of the 5th Annual ACM Symp. on Theory of Computing (STOC)*, pages 1–9. ACM, 1973.

[96] René Thiemann and Jürgen Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Alg. in Eng., Comm. & Comp.*, 16(4):229–270, 2005.

[97] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2:115–125, 1968.

[98] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[99] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Univ. Math. Lab., Cambridge, 1949. Reprinted in [74].

[100] Robert Wille, Görschwin Fey, Daniel Große, Stephan Eggersglüß, and Rolf Drechsler. Sword: A SAT like prover using word level information. In *Proc. Intl. Conf. on Very Large Scale Integration of System-on-Chip*, pages 88–93. IEEE, 2007.

[101] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. In *Proc. of FMCAD*, pages 239–246. IEEE, 2010.

[102] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[103] Yinlei Yu and Sharad Malik. Validating the result of a quantified Boolean formula (QBF) solver: theory and practice. In *Proc. of ASP-DAC*, pages 1047–1051. ACM Press, 2005.

[104] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proc. of CP*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.

Curriculum Vitae

Christoph Michael Wintersteiger

May 15, 1979	Born in Salzburg, AT

Education

1989–1995	BRG ‘Johannes Kepler’ in Graz, AT
1995–1998	BG/BRG II ‘Christian Doppler’ in Salzburg, AT
1999–2001	Studies in Telematics at Technical University Graz, AT
2001–2006	Studies in Computer Science at JK-University Linz, AT
2006–2011	Ph.D. Studies in Computer Science at ETH Zurich, CH

Employment

1998–1999	Community Service (in lieu of military service)
1996–2006	Freelance work as a network technician and programmer
2006–2010	Research Assistant at ETH Zurich, CH
2008–2009	Research Intern at Microsoft Research, USA & UK
2010	Research Intern at Microsoft Research, UK
2010	Postdoctoral Researcher at Oxford University, UK
2010–	Postdoctoral Researcher at Microsoft Research, UK

List of publications

Christoph Michael Wintersteiger

Books

- *Digitaltechnik – eine praxisnahe Einführung*
  (Digital circuits – a practical introduction)
  with A. Biere, D. Kroening, and G. Weissenbacher,
  Springer Verlag, March 2008.

- *Gustav Tauschek und seine Maschinen*
  (Gustav Tauschek and his machines)
  with M. Helfert and P. Mazuran,
  Rudolf Trauner Verlag, February 2007.

Peer-Reviewed

- *Efficiently Solving Quantified Bit-Vector Formulas*
  with Y. Hamadi, L. de Moura,
  FMCAD 2010, to appear, 2010.

- *Loopfrog – Loop Summarization for Static Analysis (Tool Abstract)*
  with D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich,
  WING 2010, to appear, 2010.

- *Termination Analysis With Compositional Transition Invariants*
  with D. Kroening, N. Sharygina, A. Tsitovich,
  CAV 2010, LNCS 6174, Springer, 2010.

- *Ranking Function Synthesis for Bit-Vector Relations*
  with B. Cook, D. Kroening, P. Rümmer,
  TACAS 2010, LNCS 6015, Springer, 2010.

- *Loopfrog: A Static Analyzer for ANSI-C Programs*
  with D. Kroening, N. Sharygina, S. Tonetta, and A. Tsitovich,
  ASE 2009, IEEE Press, 2009.

- *A Concurrent Portfolio Approach to SMT Solving*
  with Y. Hamadi, and L. de Moura,
  CAV 2009, LNCS 5643, Springer, 2009.

- *Loop Summarization using Abstract Transformers*
  with D. Kroening, N. Sharygina, S. Tonetta, and A. Tsitovich,
  ATVA 2008, LNCS 5311, Springer, 2008.

- *A First Step Towards a Unified Proof Checker for QBF*
  with T. Jussila, A. Biere, C. Sinz, and D. Kroening,
  SAT 2007, LNCS 4501, Springer, 2007.

- *Gustav Tauschek's Punchcard Accounting Machines*
  with M. Helfert,
  MEDICHI2007, OCG 220, Austrian Computer Society, 2007.

Theses

- *Crane – Eine Kryptanalyse-Umgebung*
  (Crane – A Cryptanalysis Environment)
  Masters Thesis (Diplomarbeit),
  University of Linz, Austria, February 2006.