SAT-based Software Certification

Sagar Chaki
chaki@sei.cmu.edu
Carnegie Mellon Software Engineering Institute

Abstract. We formalize a notion of witnesses for satisfaction of linear temporal logic specifications by infinite state programs. We show how such witnesses may be constructed via predicate abstraction, and validated by generating verification conditions and proving them. We propose the use of SAT-based theorem provers and resolution proofs in proving these verification conditions. In addition to yielding extremely compact proofs, a SAT-based approach overcomes several limitations of conventional theorem provers when applied to the verification of programs written in real-life programming languages. We also formalize a notion of witnesses of simulation conformance between infinite state programs and finite state machine specifications. We present algorithms to construct simulation witnesses of minimal size by solving pseudo-Boolean constraints. We present experimental results on several non-trivial benchmarks which suggest that a SAT-based approach can yield extremely compact proofs, in some cases by a factor of over 10^5, when compared to existing non-SAT-based theorem provers.

1 Introduction

There is an evident and urgent need for objective measures of confidence in the *behavior* of software obtained from untrusted sources. In general, the lack of trust in a piece of code stems from two sources: (i) the code producer, and (ii) the mechanism of delivery of the code to the consumer. Unfortunately, the vast majority of current software assurance techniques target the above sources of mistrust in isolation, but fail to account for them both.

For instance, cryptographic techniques are typically unable to say anything substantial about the run-time behavior of the program. Techniques such as sandboxing and analytic redundancy require mechanisms for run-time monitoring and appropriate responses to failure. Additionally, such approaches are inherently dynamic and unable to provide adequate levels of static correctness guarantees. Extrinsic software quality standards typically have a heavy focus on process and are usually quite subjective. Moreover, software qualities are weakly related to desired behavior, if at all.

This article presents a technique that uses *proofs* to certify software. More specifically, we certify a rich set of safety and *liveness* policies on C source code. Our approach consists of two broad stages. We first use model checking [13, 11] in conjunction with CounterExample Guided Abstraction Refinement (CEGAR) [12] and predicate abstraction [17] to verify that a C program $\mathcal{C}$ satisfies a policy $\mathcal{S}$. The policy $\mathcal{S}$ may be expressed either as a linear temporal logic (LTL) formula or a finite state machine.

Subsequently, we use information generated by the verification procedure to extract a witness Ω. We show how the witness may be used to generate a verification condition VC. We also prove that $\mathcal{C}$ respects the policy $\mathcal{S}$ iff VC is valid. The witness Ω is constructed and shipped by the code producer along with $\mathcal{C}$ and the proof P of VC. The code consumer uses Ω to reconstruct VC and verify that P truly corresponds to VC. Therefore, in our setting, the witness Ω and the proof P may together be viewed as the certificate that $\mathcal{C}$ respects $\mathcal{S}$.

While the above strategy is theoretically sound, it must overcome two key pragmatic obstacles. First, since certificates have to be transmitted and verified, they must be small and efficiently checkable. Unfortunately, proofs generated by conventional theorem-provers, such as CVC and VAMPYRE, are often prohibitively large. Second, conventional theorem provers are usually unfaithful to the semantics of C. For example, they often do not support features of integer operations such as overflow and underflow. This means that certificates generated by such theorem provers are, in general, not trustworthy. For example, the following VC is declared valid by most conventional theorem provers, including CVC and VAMPYRE: $\forall x \centerdot (x+1) > x$. However, the above statement is actually invalid according to the semantics of the C language due to the possibility of overflow.

In this article, we propose the use of Boolean satisfiability (SAT) to solve both these problems. More specifically, we translate VC to a propositional formula Φ such that VC is valid iff Φ is unsatisfiable. Therefore, a resolution refutation (proof of the unsatisfiability) of Φ serves as a proof of the validity of VC. We use the state-of-the-art SAT solver ZCHAFF [25], which also generates resolution refutations, to prove that Φ is unsatisfiable. The translation from VC to Φ is faithful to the semantics of C and therefore handles issues such as overflow.

We have implemented our proposed technique in the ComFoRT [10] reasoning framework and experimented with several non-trivial benchmarks. Our results indicate that the use of SAT leads to extremely compact (in some cases over 10^5 times smaller) proofs in comparison to conventional theorem-provers. Further details of our experiments can be found in Section 7.

We believe that this paper contributes to not just the area of software certification, but to the much broader spectrum of scientific disciplines where compact proofs are desirable. Algorithms to compress proof representations are currently a topic of active research. This article demonstrates that the use of SAT technology is a very promising idea in this context. In the rest of this paper, we omit proofs of lemmas and theorems for the sake of brevity. Detailed proofs can be found in an extended version of this paper [6].

2 Related Work

Necula and Lee [28, 30, 31] proposed PCC as a means for checkably certifying that untrusted *binaries* respect certain fundamental safety (such as memory safety) criteria. Foundational PCC [2, 18] attempts to reduce the trusted computing base of PCC to solely the foundations of mathematical logic. Bernard and Lee [5] propose a new temporal logic to express PCC policies for machine code.Non-SAT-based techniques for minimizing PCC proof sizes [29, 32] and formalizing machine code semantics [24] have also been proposed. Our work uses proofs to certify software but is applicable to safety as well as liveness specifications, and at the *source code* level.

Certifying model checkers [26, 22] emit an independently checkable certificate of correctness when a temporal logic formula is found to be satisfiable by a *finite state* model. Namjoshi [27] has proposed a two-step technique for obtaining proofs of μ-calculus specifications on *infinite-state* systems. In the first step, a proof is obtained via certifying model checking. In the second step, the proof is *lifted* via an abstraction. This approach is more general than ours as far as LTL model checking is concerned, but does not handle simulation. It also does not propose the use of SAT or provide experimental validation.

Magill et al. [23] have proposed a two-step procedure for certifying simulation conformance between an infinite-state system and a finite state machine specification. In the first step they certify that a finite-state abstraction simulates the infinite-state system. In the second step they prove simulation between the finite-state abstraction and the specification. Their approach does not cover LTL specifications, and in particular, is unable to handle liveness policies. Also, it does not propose the use of SAT.

Predicate abstraction [17] in combination with CEGAR [12] has been applied successfully by several software model checkers such as SLAM [4], BLAST [20] and MAGIC [7]. Out of these SLAM and MAGIC do not generate any proof certificates when claiming the validity of program specifications. BLAST includes a method [19] for lifting linear time safety proofs through the abstraction computed by their algorithm into a checkable proof of correctness for the original program. It does not handle liveness specifications and uses the non-SAT-based theorem prover VAMPYRE for proof generation. The use of SAT for software model checking has also been explored in the context of both sequential ANSI-C programs [14] and asynchronous concurrent Boolean programs [15]. Proving program termination via ranking functions is also a rich, and developing, research area [16, 3].

3 Preliminaries

In this section we present preliminary definitions and results. Let Act be a denumerable set of actions. We begin with the notion of labeled transition systems.

Definition 1 (LTS). *A Labeled Transition System (LTS) is a quadruple* $(S, Init, \Sigma, T)$ *where: (i)* S *is a finite set of states, (ii)* $Init \subseteq S$ *is a set of initial states, (iii)* $\Sigma \subseteq Act$ *is a finite alphabet, and (iv)* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

Given an LTS $M = (S, Init, \Sigma, T)$ we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. Also for any $s \in S$ and any $\alpha \in \Sigma$ we denote by $Succ(s, \alpha)$ the set of successors of s under α. In other words: $Succ(s, \alpha) = \{s' \mid s \xrightarrow{\alpha} s'\}$.

Linear Temporal Logic. We now define our notion of linear temporal logic (LTL). Unlike standard practice, the flavor of LTL we use is based on actions instead of propositions. This distinction is, however, inessential as far as this article is concerned. The syntax of LTL is defined by the following BNF-style grammar (where $\alpha \in \mathit{Act}$): $\phi := \alpha \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\phi_1 \mid \phi_1 \mathbf{U} \phi_2$.

The semantics of LTL is fairly standard and we do not describe it here. In fact, we do not deal with LTL specifications directly but rather via an equivalent automata-theoretic formalism called Büchi automata.

Definition 2 (Büchi Automaton). *A Büchi automaton (or simply an automaton) is 5-tuple $(S, \mathit{Init}, \Sigma, T, F)$ where: (i) S is a finite set of states, (ii) $\mathit{Init} \subseteq S$ is a set of initial states, (iii) $\Sigma \subseteq \mathit{Act}$ is a finite alphabet, (iv) $T \subseteq S \times \Sigma \times S$ is a transition relation, and (v) $F \subseteq S$ is a set of final (or accepting) states.*

As in the case of LTSs, given a Büchi Automaton $B = (S, \mathit{Init}, \Sigma, T, F)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. Also for any $s \in S$ and any $\alpha \in \Sigma$, we denote by $\mathit{Succ}(s, \alpha)$ the set $\{s' \mid s \xrightarrow{\alpha} s'\}$.

Language. A *trace* $t \in \mathit{Act}^\omega$ is an infinite sequence of actions. The language accepted by an automaton is a set of traces defined as follows. Let $B = (S, \mathit{Init}, \Sigma, T, F)$ be any automaton and $t = \langle \alpha_0, \alpha_1, \ldots \rangle$ be any trace. A *run* r of B on t is an infinite sequence of states $\langle s_0, s_1, \ldots \rangle$ such that: (i) $s_0 \in \mathit{Init}$ and (ii) $\forall i \geq 0 \centerdot s_i \xrightarrow{\alpha_i} s_{i+1}$. For any run r we write $\mathit{Inf}(r)$ to denote the set of states appearing infinitely often in r. Then a trace t is accepted by B iff there exists a run r of B on t such that $\mathit{Inf}(r) \cap F \neq \emptyset$. The language of B, denoted by $\mathcal{L}(B)$ is the set of traces accepted by B. We define the product between an LTS and an automaton in the standard manner as follows:

Definition 3 (Product Automaton). *Let $M = (S_1, \mathit{Init}_1, \Sigma_1, T_1)$ be an LTS and $B = (S_2, \mathit{Init}_2, \Sigma_2, T_2, F_2)$ be an automaton such that $\Sigma_1 = \Sigma_2$. Then the product of M and B is denoted by $M \otimes B$ and defined as the automaton $(S, \mathit{Init}, \Sigma, T, F)$ where: (i) $S = S_1 \times S_2$, (ii) $\mathit{Init} = \mathit{Init}_1 \times \mathit{Init}_2$, (iii) $\Sigma = \Sigma_1$, (iv) $F = S_1 \times F_2$ and (v) T is defined as follows: $(s_1, s_2) \xrightarrow{\alpha} (s'_1, s'_2) \iff s_1 \xrightarrow{\alpha} s'_1 \wedge s_2 \xrightarrow{\alpha} s'_2$.*

Program. We have applied our ideas to actual C programs. However, for clarity and simplicity of presentation, we use a programming language based on guarded commands. Let Var be a denumerable set of integer variables. The set of expressions Expr is defined over Var using the following operators : $+, -, \times, \div, =, <, \neg, \wedge$ and the C bit-wise operators.

Program Syntax. An assignment is a pair (v, e) where $v \in \mathit{Var}$ denotes the left-hand-side (LHS) and $e \in \mathit{Expr}$ denotes the right-hand-side (RHS). The set of assignments is denoted by Asgn. A guarded command is a triple $(\mathit{Grd}, \mathit{Evt}, \mathit{Cmd})$ where $\mathit{Grd} \in \mathit{Expr}$ is a guard, $\mathit{Evt} \in \mathit{Act}$ is an event and $\mathit{Cmd} \in \mathit{Asgn}$ is an assignment. The set of guarded commands is denoted by GrdCmd. Given a guarded command $gc = (g, e, c)$ we write $\mathit{Grd}(gc)$, $\mathit{Evt}(gc)$ and $\mathit{Cmd}(gc)$ to denote g, e and c respectively. Finally, a program is a pair (I, C) where $I \in \mathit{Expr}$ expresses constraints on the initial states of the program and $C \subseteq \mathit{GrdCmd}$ is a finite set of guarded commands.

Store. A *store* is a function $\sigma : \mathit{Var} \to \mathbb{Z}$ from variables to integers. The set of all stores is denoted by Sto. Any store σ naturally induces a function from expressions to integers: $\sigma(e)$ is the integer obtained by evaluating e under σ.

Our language has a C-like semantics as far as variables and operators are concerned. Integers are treated as 32-bit vectors. Also, the arithmetic, relational, Boolean and bit-wise operators are interpreted in a C-like manner. In particular, there is overflow and underflow, zero is treated as FALSE, while all other integers are treated as TRUE.

Definition 4 (Store Update). *Given a store σ and an assignment $a = (v, e)$ we write $a[\sigma]$ to denote the store resulting after executing a from σ. In other words, $a[\sigma]$ is the same as σ for all variables other than v, while $a[\sigma](v) = \sigma(e)$.*

Definition 5 (Satisfaction). *Given a store σ and an expression e we say that σ satisfies e iff $\sigma(e) \neq 0$. We denote the satisfaction of e by σ as $\sigma \models e$ and write $\sigma \not\models e$ to mean $\neg(\sigma \models e)$.*

In the rest of this article we use the terms formula and expression synonymously since, as we have seen, any expression e can also be viewed as a logical formula. The models of e are simply the stores satisfying e.

Program Semantics. We now define the semantics of a program *Prog* in terms of a labeled transition system. Intuitively, the states of the LTS are stores, its initial states are determined by the initial condition of *Prog*, and its transitions are determined by the guarded commands in *Prog*. Formally, let $\mathit{Prog} = (I, C)$ be a program. Then the semantics of *Prog*, denoted by $[\![\mathit{Prog}]\!]$, is an LTS $(S, \mathit{Init}, \Sigma, T)$ such that: (i) $S = \mathit{Sto}$, (ii) $\mathit{Init} = \{\sigma \mid \sigma \models I\}$, (iii) $\Sigma = \{\mathit{Evt}(gc) \mid gc \in C\}$, and (iv) $\sigma \xrightarrow{\alpha} \sigma'$ iff: $\exists gc \in C \centerdot \sigma \models \mathit{Grd}(gc) \wedge \alpha = \mathit{Evt}(gc) \wedge \sigma' = \mathit{Cmd}(gc)[\sigma]$. Given a specification as a *negated* automaton *Spec*, we say that *Prog* satisfies *Spec*, and denote this by $\mathit{Prog} \models \mathit{Spec}$, iff $\mathcal{L}([\![\mathit{Prog}]\!] \otimes \mathit{Spec}) = \emptyset$.

4 Temporal Logic Witness

In this section we present our proof framework for programs. We consider a program $\mathit{Prog} = (I, C)$. We begin with the notion of strongest postconditions. For any expression e, variable v and expression t, we denote the expression obtained by simultaneously replacing all occurrences of v in e by t as $e[v/t]$.

Definition 6 (Strongest Postcondition). *Let $\mathit{Prog} = (I, C)$ be a program, e be an expression and α be an action. Then the strongest postcondition of e w.r.t. α is denoted by $\mathcal{SP}[e]\{\alpha\}$ and defined as follows: $\mathcal{SP}[e]\{\alpha\} = \exists v' \centerdot \bigvee_{(g,\alpha,(v,t)) \in C} (g \wedge e)[v/v'] \wedge (v = t[v/v'])$.*

The concept of strongest postconditions is quite standard. In particular, the following fact about strongest postconditions is fairly well-known. Recall that a state of *Prog* is a store. Consider any expression e and any action α. Let σ and σ' be stores such that $\sigma \models e$ and $\sigma \xrightarrow{\alpha} \sigma'$. Then $\sigma' \models \mathcal{SP}[e]\{\alpha\}$. This idea is captured by the following well-known fact.

Fact 1 *Let Prog be a program and* $[\![Prog]\!] = (S, Init, \Sigma, T)$ *be its semantics. Let* e *be any expression. Then the following holds:* $\forall \sigma \in S \centerdot \forall \sigma' \in S \centerdot \forall \alpha \in \Sigma \centerdot ((\sigma \models e) \wedge (\sigma \xrightarrow{\alpha} \sigma')) \Rightarrow (\sigma' \models \mathcal{SP}[e]\{\alpha\})$.

Lemma 1. *Let* e_1, e_2 *be any expressions and* α *be any action. Then the following holds:* $(\mathcal{SP}[e_1]\{\alpha\} \vee \mathcal{SP}[e_2]\{\alpha\}) \iff \mathcal{SP}[e_1 \vee e_2]\{\alpha\}$.

We are now ready to present the formal notion of a proof of $Prog \models Spec$. Recall that our goal is to prove $\mathcal{L}([\![Prog]\!] \otimes Spec) = \emptyset$. Such a proof essentially encodes a *stratified ranking function* between $[\![Prog]\!]$ and *Spec*. Let us write $M_\otimes$ to mean $[\![Prog]\!] \otimes Spec$. Let $M_\otimes = (S_\otimes, Init_\otimes, \Sigma, T_\otimes, F_\otimes)$ and R be a finite set of integral ranks. Suppose that there exists a ranking function $\rho : S_\otimes \rightarrow R$ such that the following holds:

- **(RANK1)** $Init_\otimes \subseteq Domain(\rho)$, i.e., all initial states of $M_\otimes$ have a rank.
- **(RANK2)** $\forall s \xrightarrow{\alpha} s' \centerdot s \notin F_\otimes \Rightarrow \rho(s) \geq \rho(s')$.
- **(RANK3)** $\forall s \xrightarrow{\alpha} s' \centerdot s \in F_\otimes \Rightarrow \rho(s) > \rho(s')$.

Then there is no infinite path of $M_\otimes$ that visits an accepting state infinitely often, i.e., $\mathcal{L}(M_\otimes) = \emptyset$. We use a witness to encode a ranking function. We also use appropriate side-conditions to ensure that the ranking function satisfies the three conditions mentioned above. We now state this formally:

Theorem 1. *Let* $Prog = (I, C)$ *be a program and* $Spec = (S, Init, \Sigma, T, F)$ *be a specification automaton. Let* R *be a finite set of integral ranks. Suppose that there exists a function* $\Omega : S \times R \rightarrow Expr$ *that satisfies the following four conditions:*
(C1) $\forall s \in S \centerdot \forall r \in R \centerdot \forall r' \in R \centerdot r \neq r' \Rightarrow \neg(\Omega(s, r) \wedge \Omega(s, r'))$
(C2) $\forall s \in Init \centerdot I \Rightarrow \bigvee_{r \in R} \Omega(s, r)$
(C3) $\forall s \in S \setminus F \centerdot \forall \alpha \centerdot \forall r \in R \centerdot \forall s' \in Succ(s, \alpha) \centerdot \mathcal{SP}[\Omega(s, r)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s', r')$
(C4) $\forall s \in F \centerdot \forall \alpha \centerdot \forall r \in R \centerdot \forall s' \in Succ(s, \alpha) \centerdot \mathcal{SP}[\Omega(s, r)]\{\alpha\} \Rightarrow \bigvee_{r' < r} \Omega(s', r')$
Then $[\![Prog]\!] \models Spec$ *and we say that* Ω *is a witness to* $[\![Prog]\!] \models Spec$.

Suppose we are given *Prog*, $Spec = (S, Init, \Sigma, T)$ and a candidate witness Ω over a set of ranks R. Since S, Σ and R are all finite, it is straightforward to generate a formula equivalent to the conditions **C1** $-$ **C4** enumerated in Theorem 1. We call such a formula our *verification condition* and denote it by $VC(Prog, Spec, \Omega)$. In essence, on account of Theorem 1, a valid proof of $VC(Prog, Spec, \Omega)$ is also a valid proof of $Prog \models Spec$.

Theorem 1 is useful in checking the validity of a proposed witness Ω. However, it yields no technique to construct such a Ω. In this section we present a procedure called predicate abstraction. In the next section we show how to construct a valid witness using predicate abstraction. More specifically, if our procedure actually results in a witness Ω, then Ω is guaranteed to be valid. In other words, the verification condition $VC(\mathit{Prog}, \mathit{Spec}, \Omega)$ is guaranteed to be a valid formula. We begin with some preliminary definitions:

Definition 7 (Predicate). *A predicate is simply an expression. Let $\mathcal{P}$ be a finite set of predicates. A valuation of $\mathcal{P}$ is a function from $\mathcal{P}$ to $\{\text{TRUE}, \text{FALSE}\}$. The set of all valuations of $\mathcal{P}$ is denoted by $\mathcal{V}(\mathcal{P})$. Given a valuation $V \in \mathcal{V}(\mathcal{P})$ of $\mathcal{P}$, the concretization of $\mathcal{P}$ w.r.t. V is denoted by $\gamma^{\mathcal{P}}(V)$ and is the expression defined as follows: $\gamma^{\mathcal{P}}(V) = \bigwedge_{p \in \mathcal{P}} p^{V(p)}$, where for any predicate p, we have $p^{\text{TRUE}} = p$ and $p^{\text{FALSE}} = \neg p$.*

In this article we only consider finite sets of predicates. We write $\gamma(V)$ to mean $\gamma^{\mathcal{P}}(V)$ when $\mathcal{P}$ is clear from context. The notion of concretization presented above means that any valuation V can also be thought of as the expression $\gamma(V)$. This leads naturally to the notion of consistency between valuations and expressions and between two valuations.

Definition 8 (Consistency). *Let V be a valuation of a set of predicates $\mathcal{P}$ and e be an expression. We say that V is consistent with e, and denote this by $V \Vdash e$, iff the expression $\gamma(V) \Rightarrow \neg e$ is invalid. In other words: $V \Vdash e \iff \exists \sigma \in \mathit{Sto} \centerdot \sigma \models \gamma(V) \wedge \sigma \models e$. Equivalently, $\neg(V \Vdash e)$ iff the expression $\gamma(V) \Rightarrow \neg e$ is valid.*

Consistency essentially means that a valuation and an expression are not mutually exclusive. We now define weakest preconditions, a concept closely related to strongest postconditions. Recall that for any expression e, variable v and expression t, we denote the expression obtained by simultaneously replacing all occurrences of v in e by t as $e[v/t]$.

Definition 9 (Weakest Precondition). *Let $\mathit{Prog} = (I, C)$ be a program, e be an expression and α be an action. Then the weakest precondition of e w.r.t. α is denoted by $\mathcal{WP}[e]\{\alpha\}$ and defined as: $\mathcal{WP}[e]\{\alpha\} = \bigvee_{(g,\alpha,(v,t)) \in C} g \wedge e[v/t]$.*

The relationship between strongest postconditions and weakest preconditions is expressed formally by the following lemma.

Lemma 2. *Let e, e' be expressions and α be an action. Then the following holds: $(e \Rightarrow \neg \mathcal{WP}[e']\{\alpha\}) \Rightarrow (\mathcal{SP}[e]\{\alpha\} \Rightarrow \neg e')$.*

Predicate Abstraction. Let $\mathit{Prog} = (I, C)$ be a program and $\mathcal{P}$ be a set of predicates. Let $[\![\mathit{Prog}]\!] = (S, \mathit{Init}, \Sigma, T)$ be the semantics of Prog. Then the predicate abstraction of Prog w.r.t. $\mathcal{P}$ is denoted by $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}}$ and is defined as an LTS $(\widehat{S}, \widehat{\mathit{Init}}, \widehat{\Sigma}, \widehat{T})$ where: (i) $\widehat{S} = \mathcal{V}(\mathcal{P})$: the states are the valuations of $\mathcal{P}$, (ii) $\widehat{\mathit{Init}} = \{V \in \mathcal{V}(\mathcal{P}) \mid V \Vdash I\}$, (iii) $\widehat{\Sigma} = \Sigma$, and (iv) $\widehat{T}$ is defined as follows: $V \xrightarrow{\alpha} V' \iff V \Vdash \mathcal{WP}[\gamma(V')]\{\alpha\}$.

Predicate abstraction enables us to create finite LTS abstractions of our infinite state programs. More importantly, it can be automated. Given Prog and $\mathcal{P}$ it is easy to construct $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}}$ from the definition given above. In order to check for consistency we use an automated theorem-prover. More specifically, suppose we want to check if $V \Vdash e$. Then, in accordance with Definition 8, we check for the validity of $\gamma(V) \Rightarrow \neg e$ using a (sound) theorem prover. We assume $\neg(V \Vdash e)$ iff the theorem says that $\gamma(V) \Rightarrow \neg e$ is valid.

Generating LTL Witnesses. We now present an algorithm **WitGen** for constructing a valid witness to $[\![\mathit{Prog}]\!] \models \mathit{Spec}$. The input to **WitGen** is: (i) a set of predicates $\mathcal{P}$ such that $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \models \mathit{Spec}$, and (ii) a ranking function ρ from the states of $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \otimes \mathit{Spec}$ to a finite set of ranks R that obeys conditions **RANK1** – **RANK3** given in Section 4. We defer the question as to how such a set of predicates $\mathcal{P}$ and ranking function ρ may be constructed till later. The output of **WitGen** is a valid witness Ω. The following theorem conveys the key ideas behind our algorithm.

Theorem 2 (Valid Witness). *Let* $\mathit{Prog} = (I, C)$ *be a program,* $\mathit{Spec} = (S, \mathit{Init}, \Sigma, T, F)$ *be a finite specification automaton and* $\mathcal{P}$ *be a set of predicates such that* $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \models \mathit{Spec}$. *Let* $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{\mathit{Init}}, \widehat{\Sigma}, \widehat{T})$. *Let* R *be a finite set of integral ranks and* $\rho : \mathcal{V}(\mathcal{P}) \times S \to R$ *be a ranking function that obeys conditions* **RANK1** – **RANK3** *given in Section 4. Now consider the witness* $\Omega : S \times R \to \mathit{Expr}$ *defined as follows:* $\Omega(s, r) = \bigvee_{V : \rho(V,s) = r} \gamma(V)$. *Then* Ω *is a valid witness to* $[\![\mathit{Prog}]\!] \models \mathit{Spec}$.

Getting Predicates and Ranking Functions. Theorem 2 immediately leads to an algorithm **WitGen** to construct a valid witness Ω to $\mathit{Prog} \models \mathit{Spec}$. However, **WitGen** requires as input an appropriate set of predicates $\mathcal{P}$ such that $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \models \mathit{Spec}$, as well as a ranking function ρ satisfying the conditions mentioned in Theorem 2. A suitable $\mathcal{P}$ may be constructed by combining predicate abstraction with CEGAR. Full details of such a procedure can be found elsewhere [9]. Due to the fundamental undecidability of the problem, such an approach is not always guaranteed to terminate. However, CEGAR-based techniques have been reported to be quite successful [4, 20, 7] in software verification in recent times.

Generating the Ranking Function. Once an appropriate set of predicates $\mathcal{P}$ has been found by the above procedure, we have to construct a ranking function ρ. More precisely, suppose that $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{\mathit{Init}}, \widehat{\Sigma}, \widehat{T})$ and $\mathit{Spec} = (S, \mathit{Init}, \Sigma, T, F)$. Then we have to construct: (i) a finite set of integral ranks R, and (ii) a ranking function $\rho : \mathcal{V}(\mathcal{P}) \times S \to R$ that obeys conditions **RANK1** – **RANK3** given in Section 4. We now give an algorithm to achieve these two goals.

Let us denote $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \otimes \mathit{Spec}$ by $M_\otimes$ and let $M_\otimes = (S_\otimes, \mathit{Init}_\otimes, \Sigma, T_\otimes, F_\otimes)$. Without loss of generality we assume that both $S_\otimes$ and $F_\otimes$ only contain the states of $M_\otimes$ that are reachable from $\mathit{Init}_\otimes$ via the transition relation. Our ranking function is defined on only $S_\otimes$, and undefined for unreachable states of $M_\otimes$.

First, we note that $M_\otimes$ can be viewed as a directed graph $G_\otimes = (N, E)$ such that: $(N = S_\otimes) \bigwedge (E = \{(s, s') \mid \exists \alpha \in \Sigma \centerdot s \xrightarrow{\alpha} s'\})$. Given any two nodes s and s' we say that $s \rightsquigarrow s'$ iff there is a path from s to s' in G. In other words,

$s \rightsquigarrow s'$ iff there exists a finite *non-empty* sequence of states $s_1, s_2, \ldots, s_k$ such that: $(s = s_1) \bigwedge (s' = s_k) \bigwedge (\forall i \in \{1, \ldots, k-1\} \centerdot (s_i, s_{i+1}) \in E)$. A strongly connected component (SCC) of $G_\otimes$ is a set of nodes $X \subseteq N$ such that: $\forall s \in X \centerdot \forall s' \in X \centerdot s \rightsquigarrow s'$. A node of $G_\otimes$ that does not belong to any SCC is called a *finitary* node. It is evident that a node n is finitary iff for every run x of $M_\otimes$ we have $n \notin \mathit{Inf}(x)$. We also know that $\{\!\{\mathit{Prog}\}\!\}^{\mathcal{P}} \models \mathit{Spec}$ and hence $\mathcal{L}(M_\otimes) = \emptyset$. This means that every accepting state $s \in F_\otimes$ must be finitary.

It is also well known that every directed graph G induces a directed acyclic graph G^{SCC}. The nodes of G^{SCC} are the maximal strongly connected components and the finitary nodes of G while its edges are induced by those of G. Let $G_\otimes^{SCC}$ be the directed acyclic graph induced by $G_\otimes$. Let $\mathcal{O} = \langle n_1, n_2, \ldots, n_k \rangle$ be a topological ordering of the nodes of $G_\otimes^{SCC}$ such that if $n_i \rightsquigarrow n_j$, then n_j appears before n_i in $\mathcal{O}$. We now fix our set of ranks R to be $\{1, 2, \ldots, k\}$ where $k = |\mathcal{O}|$. We first define a ranking function ρ^{SCC} for the nodes of $G_\otimes^{SCC}$ as follows: $\rho^{SCC}(n) = i$ iff $n = n_i$ according to the ordering $\mathcal{O}$. We then use ρ^{SCC} to define a ranking function ρ for $G_\otimes$ as follows:

– If n is a finitary node then it is also a node of $G_\otimes^{SCC}$. Then $\rho(n) = \rho^{SCC}(n)$.
– Otherwise n belongs to an unique maximal SCC n^{SCC} which is a node of $G_\otimes^{SCC}$. In this case $\rho(n) = \rho^{SCC}(n^{SCC})$.

We now show that ρ satisfies conditions **RANK1** − **RANK3** given in Section 4. Condition **RANK1** holds because $\mathit{Init}_\otimes \subseteq S_\otimes = \mathit{Domain}(\rho)$. For condition **RANK2**, consider any transition $s \xrightarrow{\alpha} s'$ of $M_\otimes$ such that $s \notin F_\otimes$. Now since $s \rightsquigarrow s'$ we have $\rho(s) \geq \rho(s')$ which is precisely **RANK2**. For condition **RANK3**, consider any transition $s \xrightarrow{\alpha} s'$ of $M_\otimes$ such that $s \in F_\otimes$. Recall that in this case s must be a finitary node. Hence $\rho(s) \neq \rho(s')$. Since $s \rightsquigarrow s'$ we have $\rho(s) > \rho(s')$ which is precisely **RANK3**.

The use of ranking functions for proofs of liveness properties is well studied and ours is but another instance of this methodology. The use, and limitations, of CEGAR for generating appropriate predicates is orthogonal to the witness construction procedure. In practice, any oracle capable of providing a suitable set of predicates can be substituted for CEGAR. For instance, some of the predicates can be manually supplied and the remaining predicates may be constructed automatically.

5 SAT-based Certificates

Suppose we are given a program *Prog*, a specification *Spec* and a candidate witness Ω. We wish to check the validity of Ω. To this end we construct the verification condition $\mathit{VC} = \mathit{VC}(\mathit{Prog}, \mathit{Spec}, \Omega)$ and prove that VC is valid. One way to achieve this goal is to pass VC as a query to an existing proof-generating automated theorem-prover such as CVC or VAMPYRE. However, there are at least two shortcomings of this approach:

First, most theorem provers treat integers, as well as operations on integers, in a manner that is incompatible with the semantics of our programming language. For example, our language defines integers to be 32-bit vectors and operations such as addition and multiplication are defined in accordance with two's-complement arithmetic. In contrast, for most theorem provers, integers have an infinite domain and operations on them are the ones we learn in primary school. An important consequence of this discrepancy is that certificates generated by conventional theorem provers may be untrustworthy for our purposes. For example, the following verification condition is declared valid by most conventional theorem provers, including CVC and VAMPYRE: $\forall x \centerdot (x+1) > x$. However, the above statement is actually invalid according to our language semantics due to the possibility of overflow.

In addition, the proofs generated by such theorem provers are usually quite large (cf. Figure 2). We propose the use of a SAT-based proof-generating decision procedure to overcome both these hurdles. Recall that the verification conditions we are required to prove are essentially expressions. Given a verification condition VC, we check its validity as follows:

1. We translate VC to a SAT formula Φ in conjunctive normal form such that VC is valid iff Φ is unsatisfiable. In essence Φ represents the negation of VC.
2. We check for the satisfiability of Φ using a SAT solver. If Φ is found to be satisfiable then VC is invalid. Otherwise, Φ is unsatisfiable and therefore VC is valid. In such a case our SAT solver also emits a resolution[1] proof P that refutes Φ. We use P as the proof of validity of VC.

In our implementation, we use the CPROVER [21] tool to perform Step 1 above. Step 2 is performed by the state-of-the-art SAT solver ZCHAFF [25] which is capable of generating resolution-based refutation proofs [34]. The ZCHAFF distribution also comes with a proof checker which we use to verify the correctness of the proofs emitted by ZCHAFF as a sanity-check. We discuss our experimental results in detail in Section 7. We note here that in almost all cases, SAT-based proofs are over 100 times (in one case over 10^5 times) more compact than those generated by CVC and VAMPYRE. Of course, our proofs are additionally faithful to the semantics of our programming language.

It is important to understand how our approach addresses the two shortcomings of conventional theorem provers presented at the beginning of this section. The first problem regarding language semantics is handled by the translation from VC to Φ in Step 1 above. Of course, the translator itself now becomes part of our trusted computing base. However, we believe that such a decision is amply justified by the resulting benefits.

The second difficulty with large proof sizes is mitigated by the fact that a Φ generated from real-life programs and specifications often has an extremely compact resolution refutation. Intuitively, if a program is correct, it is usually correct because of some simple reason. In practice, this results in Φ having a much smaller unsatisfiable core C. In essence, C is a subset of the clauses in Φ that is itself unsatisfiable. Since Φ is in CNF form, it is possible to refute Φ by simply refuting C. State-of-the-art SAT solvers, such as ZCHAFF, leverage this

[1] Resolution is a sound and complete inference rule for refuting propositional formulas.

idea by first computing a small unsatisfiable core of the target formula and then generating a refutation for only the core. Section 7 contains more details about the kind of compression we are typically able to obtain by using the unsatisfiable core.

Finally, we note that the use of SAT guarantees trustworthiness of the generated certificate even if we use a non-SAT-based theorem prover, such as SIMPLIFY [33], for predicate abstraction. This enables us to use fast, but potentially unfaithful, theorem provers during the verification stage and still remain faithful to C semantics as far as certification is concerned.

6 Simulation

While LTL allows us to reason about both safety and liveness properties, it is nevertheless restricted to a purely linear notion of time. Simulation enables us to reason about branching time properties of programs since it preserves all ACTL* specifications.

Definition 10 (Simulation). *Let $M_1 = (S_1, Init_1, \Sigma, T_1)$ and $M_2 = (S_2, Init_2, \Sigma, T_2)$ be two LTSs. Note that M_1 and M_2 have the same alphabet. A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be a simulation relation if it satisfies the following condition:* (**SIM**) $\forall s_1 \in S_1 \centerdot \forall s_1' \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \in \Sigma \centerdot (s_1, s_2) \in \mathcal{R} \wedge s_1 \xrightarrow{\alpha} s_1' \Rightarrow \exists s_2' \in S_2 \centerdot s_2 \xrightarrow{\alpha} s_2' \wedge (s_1', s_2') \in \mathcal{R}$. *We say that M_1 is simulated by M_2, and denote this by $M_1 \preccurlyeq M_2$, iff there exists a simulation relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $\forall s_1 \in Init_1 \centerdot \exists s_2 \in Init_2 \centerdot (s_1, s_2) \in \mathcal{R}$.*

Simulation Witness. We are now ready to present the formal notion of a proof of *Prog* $\preccurlyeq$ *Spec*. Such a proof essentially encodes a simulation relation between *Prog* and *Spec*. The idea is to use a mapping Ω from states of *Spec* to expressions such that for any state s of *Spec*, $\Omega(s)$ is satisfied by those states of *Prog* that are simulated by *Spec*. We now state this formally:

Theorem 3. *Let $Prog = (I, C)$ be a program and $Spec = (S, Init, \Sigma, T)$ be a finite LTS. Suppose that there exists a function $\Omega : S \to Expr$ that satisfies the following two conditions:* **(D1)** $I \Rightarrow \bigvee_{s \in Init} \Omega(s)$ *and* **(D2)** $\forall s \in S \centerdot \forall \alpha \in \Sigma \centerdot \mathcal{SP}[\Omega(s)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s')$. *Then $[\![Prog]\!] \preccurlyeq Spec$ and we say that Ω is a witness to $[\![Prog]\!] \preccurlyeq Spec$.*

Suppose we are given *Prog*, $Spec = (S, Init, \Sigma, T)$ and a candidate witness Ω. Since both S and Σ are finite, it is straightforward to generate a formula equivalent to the conditions $\mathbf{D1} - \mathbf{D2}$ enumerated in Theorem 3. We call such a formula our *verification condition* and denote it by $VC(Prog, Spec, \Omega)$. In essence, on account of Theorem 3, a valid proof of $VC(Prog, Spec, \Omega)$ is also a valid proof of *Prog* $\preccurlyeq$ *Spec*.

Generating Simulation Witnesses. We now present an algorithm **WitGenSimul** for constructing a valid witness to $[\![Prog]\!] \preccurlyeq Spec$. The input to **WitGenSimul** is a set of predicates $\mathcal{P}$ such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$, and a simulation relation $\mathcal{R}$ between the states of $\{\!\!\{ Prog \}\!\!\}^{\mathcal{P}}$ and the states of *Spec*. We defer the question as to how such a set of predicates $\mathcal{P}$ and simulation relation $\mathcal{R}$ may be constructed till later. The output of **WitGenSimul** is a valid witness Ω. The following theorem conveys the key ideas behind our algorithm.

Theorem 4 (Valid Witness). *Let* $Prog = (I, C)$ *be a program,* $Spec = (S, Init, \Sigma, T)$ *be a finite LTS and* $\mathcal{P}$ *be a set of predicates such that* $\{\!\!\{ Prog \}\!\!\}^{\mathcal{P}} \preccurlyeq Spec$. *Let* $\{\!\!\{ Prog \}\!\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{Init}, \widehat{\Sigma}, \widehat{T})$ *and* $\mathcal{R} \subseteq \mathcal{V}(\mathcal{P}) \times S$ *be a simulation relation such that:* (**A1**) $\quad \forall V \in \widehat{Init} \centerdot \exists s \in Init \centerdot (V, s) \in \mathcal{R}$. *Let us also define a function* $\theta : S \to 2^{\mathcal{V}(\mathcal{P})}$ *as follows:* (**A2**) $\quad \forall s \in S \centerdot \theta(s) = \{V \mid (V, s) \in \mathcal{R}\}$. *Now consider the witness* $\Omega : S \to Expr$ *defined as follows:* (**A3**) $\quad \forall s \in S \centerdot \Omega(s) = \bigvee_{V \in \theta(s)} \gamma(V)$. *Then* Ω *is a valid witness to* $[\![Prog]\!] \preccurlyeq Spec$.

Getting Simulation Predicates. Theorem 4 immediately leads to an algorithm **WitGenSimul** to construct a valid witness Ω to $Prog \preccurlyeq Spec$. However, **WitGenSimul** requires as input an appropriate set of predicates $\mathcal{P}$ such that $\{\!\!\{ Prog \}\!\!\}^{\mathcal{P}} \preccurlyeq Spec$. As in the case of LTL model checking, such a $\mathcal{P}$ may be constructed by combining predicate abstraction with CEGAR. Full details of such a procedure can be found elsewhere [8]. As in the case of LTL, due to the fundamental undecidability of the problem, such an approach is not always guaranteed to terminate, but has been found to be quite effective in practice.

Witness Minimization. It is clear from Theorem 4 that the size of witnesses and proofs generated by **WitGenSimul** is directly related to the size of the simulation relation $\mathcal{R}$ between $\{\!\!\{ Prog \}\!\!\}^{\mathcal{P}}$ and *Spec*. In this section we describe an algorithm to construct a *minimal* simulation relation between two finite LTSs if such a relation exists. Clearly, such an algorithm can be used to construct an $\mathcal{R}$ of minimal size which would in turn lead to a witness Ω of small size.

Our algorithm relies on a well-known technique [7] to check for simulation between finite LTSs using satisfiability for weakly negated HORNSAT formulas. More specifically suppose we are given two finite LTSs $M_1 = (S_1, Init_1, \Sigma, T_1)$ and $M_2 = (S_2, Init_2, \Sigma, T_2)$. Then one can construct a propositional CNF formula Ψ such that the set of variables appearing in Ψ is $S_1 \times S_2$. Intuitively, a variable (s_1, s_2) stands for the proposition that state s_1 can be simulated by state s_2.

The clauses of Ψ encode constraints imposed by a simulation relation and are constructed as follows. For each $s_1 \in S_1$, each $s_2 \in S_2$, each $\alpha \in \Sigma$, and each $s'_1 \in Succ(s_1, \alpha)$ we add the following clause to Ψ: $(s_1, s_2) \Rightarrow \bigvee_{s'_2 \in Succ(s_2, \alpha)} (s'_1, s'_2)$. Intuitively the above clause expresses the requirement that for s_2 to simulate s_1, at least one α-successor of s_2 must simulate s'_1. Also, for each $s_1 \in Init_1$ we add the following clause to Ψ: $\bigvee_{s_2 \in Init_2} (s_1, s_2)$. These clauses assert that every initial state of M_1 must be simulated by some initial state of M_2. Now, Ψ has the following simple property. Let X be any satisfying assignment of Ψ and for any variable $v = (s_1, s_2)$ let us write $X(s_1, s_2)$ to mean the Boolean value assigned to v by X. Then the relation $\mathcal{R} = \{(s_1, s_2) \mid X(s_1, s_2) = \textsc{true}\}$ is a simulation relation between M_1 and M_2.

Therefore, we can construct a minimal simulation between M_1 and M_2 by constructing Ψ and then looking for a satisfying assignment X such that the number of variables assigned TRUE by X is as small as possible. This can be achieved by using a solver for pseudo-Boolean formulas [1]. A pseudo-Boolean formula is essentially a propositional formula coupled with an arithmetic constraint over the propositional variables (where TRUE is treated as one and FALSE as zero). More specifically, recall that the set of variables of Ψ is $S_1 \times S_2$. We thus solve for Ψ along with the constraint that the following sum be minimized: $\Upsilon = \sum_{s \in S_1 \times S_2} s$. We then construct a minimal simulation relation using any satisfying assignment to Ψ that also minimizes Υ.

Hardness of finding minimal simulation relations. One may complain that solving pseudo-Boolean formula satisfiability (an NP-complete problem) to verify simulation (for which polynomial time algorithms exist) is overkill. However, the use of a pseudo-Boolean solver is justified by the fact that finding a *minimal* simulation between two finite LTSs is actually an NP-hard problem.

We now prove this claim by reducing *sub-graph isomorphism*, a well-known NP-complete problem, to the problem of finding a minimal simulation relation between two LTSs. In the rest of this section, whenever we mention a simulation relation between two LTSs M_1 and M_2 we also tacitly assume that every initial state of M_1 is simulated by some initial state of M_2.

Definition 11 (Graph). *An undirected graph is a pair (V, E) where V is a set of vertices and $E \subseteq V \times V$ is a symmetric irreflexive relation denoting edges.*

Definition 12 (Subgraph Isomorphism). *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $|V_1| < |V_2|$, we say that G_1 is sub-graph isomorphic to G_2 iff there exists an injection $\mu : V_1 \rightarrow V_2$ that obeys the following condition: $\forall v \in V_1 \centerdot \forall v' \in V_1 \centerdot (v, v') \in E_1 \iff (\mu(v), \mu(v')) \in E_2$.*

Note that we do not allow self-loops in graphs. It is well-known that given two arbitrary graphs G_1 and G_2, the problem of deciding whether G_1 is sub-graph isomorphic to G_2 is NP-complete. We now show that this problem has a log-space reduction to the problem of finding a minimal simulation relation between two LTSs. In essence, from G_1 and G_2, we construct two LTSs M_1 and M_2 such that G_1 is sub-graph isomorphic to G_2 iff a minimal simulation relation between M_1 and M_2 has the same size as G_1.

Recall that $G_1 = (V_1, E_1)$. We construct $M_1 = (S_1, \mathit{Init}_1, \Sigma, T_1)$ as follows: (i) the states of M_1 are exactly the vertices of V_1, i.e., $S_1 = V_1$, (ii) all states of M_1 are initial, i.e., $\mathit{Init}_1 = S_1$, (iii) M_1 has two actions a and b, i.e., $\Sigma = \{a, b\}$, and (iv) the transitions T_1 of M_1 are set up as follows: (i) for each $(v, v') \in E_1$ we add $v \xrightarrow{a} v'$ and $v' \xrightarrow{a} v$ to T_1, and (ii) for each $(v, v') \notin E_1$ we add $v \xrightarrow{b} v'$ and $v' \xrightarrow{b} v$ to T_1. The LTS M_2 is constructed from graph G_2 in an analogous manner. As an example, Figure 1 shows two graphs G_1 and G_2 as well as the LTSs M_1 and M_2 constructed from them. Note that M_1 and M_2 can be constructed using logarithmic additional space. Now our NP-hardness reduction is completed by the following theorem.

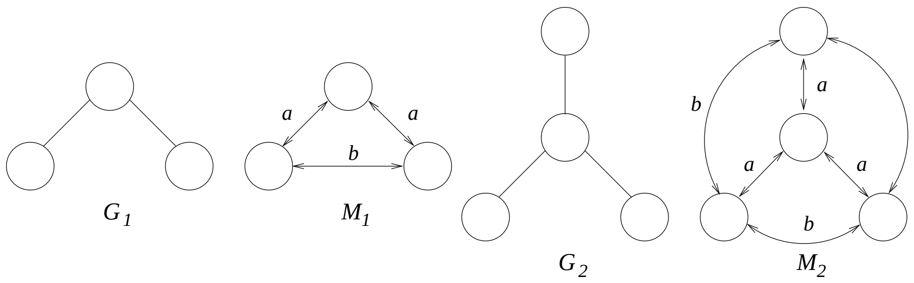

Fig. 1. Example graphs and LTSs constructed from them. A bi-directional arrow between two states represents a pair of transitions – one from each state to the other.

Theorem 5. *Let n be the number of states of M_1, i.e., $n = |S_1|$. Then G_1 is sub-graph isomorphic to G_2 iff a minimal simulation relation between M_1 and M_2 has n elements.*

7 Experimental Results

We implemented our techniques in ComFoRT [10] and experimented with a set of Linux and Windows NT device drivers, OpenSSL, and the Micro-C operating system. All our experiments were carried out on a dual Intel Xeon 2.4 GHz machine with 4 GB RAM and running Redhat 9. Our results are summarized in Figure 2. The Linux device drivers were obtained from kernel 2.6.11.10. We checked that the drivers obey the following conventions with `spin_lock` and `spin_unlock`: (i) locks must be acquired and released alternately beginning with an acquire (*safe*), and (ii) every acquire must be eventually followed by a release (*live*). The Windows drivers are instrumented so that an `ERROR` location is reached if any illegal behavior is executed. We certified that `ERROR` is unreachable for all the drivers we experimented with. For OpenSSL (version 0.9.6c) we certified that the initial handshake between a server and a client obeys the protocol specified in the SSL 3.0 specification. For Micro-C (version 2.72) we certified that the calls to `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` obey the two locking conventions mentioned above.

In almost all cases, SAT-based proofs are over 100 times more compact than those generated by CVC and VAMPYRE. In one instance – *tlan.c (live)* – the improvement is by a factor of more than 10^5. We also find that an important reason for such improvement is that the UNSAT-cores are much smaller (by over two to three orders of magnitude) than the actual SAT formulas. Upon closer inspection, we discovered that this is due to the simplicity of the verification conditions (VCs). For instance, the device drivers satisfy the locking conventions because of local coding conventions (every procedure with a lock has a matching unlock). In practice, this results in very simple VCs. Proofs generated by CVC and VAMPYRE suffer from redundancies and inefficient encodings and therefore turn out to be large even for such simple formulas. In contrast, SAT formulas generated from these simple VCs are characterized by small unsatisfiable cores.

We note that the total size of the certificate is usually dominated by the size of the witness. Finally, we find that certificates for liveness policies tend to be larger than those for the corresponding safety policies. This is due to the additional information required to encode the ranking function, which is considerably more complex for liveness specifications.

Name	*LOC*	*CVC*	*Vampyre*	*SAT*	*Cert*	*Core*	*Improve*
ide.c (safe)	7428	80720	×	**100**	**703**	>2000	807
ide.c (live)	7428	82653	×	**100**	**1319**	>2000	827
tlan.c (safe)	6523	11145980	×	**517**	**4663**	>200	21559
tlan.c (live)	6523	90155057	×	**572**	**74281**	>200	157614
aha152x.c (safe)	10069	247435	×	**210**	**2102**	>1500	1178
aha152x.c (live)	10069	247718	×	**210**	**3968**	>1500	1180
synclink.c (safe)	17104	9822	×	**53**	**185**	>500	185
synclink.c (live)	17104	9862	×	**53**	**327**	>500	186
hooks.c (safe)	30923	597642	×	**369**	**2004**	>1500	1629
hooks.c (live)	30923	601175	×	**368**	**3102**	>1500	1624
cdaudio.c (safe)	17798	248915	156787*	**209**	**2006**	>1000	750
diskperf.c (safe)	4824	117172	×	**106**	**955**	>2500	1105
floppy.c (safe)	17386	451085	60129*	**318**	**2595**	>3000	189
kbfiltr.c (safe)	12131	56682	7619*	**51**	**528**	>2500	149
parclass.c (safe)	26623	460973	×	**262**	**2156**	>4500	1759
parport.c (safe)	61781	2278120	102967*	**529**	**3568**	>5000	195
SSL-srvr (simul)	2483	1287290	19916	**261**	**1055**	>150	76
SSL-clnt (simul)	2484	189401	27189	**155**	**740**	>200	175
Micro-C (safe)	6272	416930	118162	**262**	**2694**	>5500	451
Micro-C (live)	6272	435450	×	**263**	**7571**	>5500	1656

Fig. 2. Comparison between cvc, vampyre and SAT-based proof generation. A × indicates that results are not available. Best figures are highlighted. *LOC* = lines of code. *CVC*, *Vampyre* and *SAT* = proof size in bytes (after compressing with the `gzip` utility) with cvc, vampyre and SAT. cvc statistics obtained via ComFoRT, blast statistics obtained from version 2.0 or existing publication [19] (indicated by *). *Cert* = gzipped certificate (i.e., witness + proof of the verification condition) size with SAT. *Core* = factor by which the unsatisfiable core is smaller than the original SAT formula. *Improve* = factor by which SAT-based proofs are smaller than nearest other proofs.

Acknowledgmenet. We are grateful to Stephen Magill, Aleksandar Nanevski, Peter Lee and Edmund Clarke for insight on PCC and model checking. We also thank Rupak Majumdar and Ranjit Jhala for providing us with the Windows driver benchmarks, and Anubhav Gupta for advice on zchaff.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Proc. of SAT*, 2002.
2. A. Appel. Foundational proof-carrying code. In *Proc. of LICS*, 2001.

3. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Proc. of VMCAI*, 2005.
4. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN*, 2001.
5. A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *CADE*, 2002.
6. S. Chaki. SAT-based Software Certification. Technical report CMU/SEI-2006-TN-004, Carnegie Mellon Software Engineering Institute, Pittsburgh, USA, 2006.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE TSE*, 30(6):388–402, 2004.
8. S. Chaki, E. Clarke, S. Jha, and H. Veith. An iterative framework for simulation conformance. *Journal of Logic and Computation*, 15(4), 2005.
9. S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Proc. of IFM*, 2004.
10. S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. The ComFoRT reasoning framework. In *Proc. of CAV*, 2005.
11. E. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proceedings of WLP*, 1981.
12. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.
13. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
14. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, 2004.
15. B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous boolean programs. In *Proc. of SPIN*, 2005.
16. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Proc. of SAS*, 2005.
17. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*.
18. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. of LICS*, 2002.
19. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of CAV*, 2002.
20. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *POPL'02*.
21. D. Kroening. Application specific higher order logic theorem proving. *VERIFY'02*.
22. O. Kupferman and M. Vardi. From complementation to certification. *TACAS'04*.
23. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Simulation-based safety proofs by MAGIC. In preparation.
24. N. Michael and A. Appel. Machine instruction syntax and semantics in higher order logic. In *Proc. of CADE*, 2000.
25. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC*, 2001.
26. K. Namjoshi. Certifying model checkers. In *Proc. of CAV'01*.
27. K. Namjoshi. Lifting temporal proofs through abstractions. In *VMCAI'03*.
28. G. Necula. Proof-carrying code. In *POPL'97*.
29. G. Necula and P Lee. Efficient representation and validation of proofs. In *LICS'98*.
30. G. Necula and P. Lee. Safe kernel extensions without run-time checking. *OSDI'96*.
31. G. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Proc. of Mobile Agents and Security*, 1998.
32. G. Necula and S. P. Rahul. Oracle-based checking of untrusted software. *POPL'01*.
33. G. Nelson. *Techniques for Program Verification*. PhD thesis, 1980.
34. L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE'03*.