BMC'05

Third International Workshop on Bounded Model Checking

Preliminary Proceedings

July 11, 2005

Edinburgh, Scotland

Preface

These are the *preliminary proceedings* of the third international workshop on Bounded Model Checking (BMC'05) that was held on July 11th, 2005 in Edinburgh, Scotland, UK. The final proceedings will be published in Electronic Notes in Theoretical Computer Science (ENTCS), together with other Computer Aided Verification (CAV'05) workshops. Out of 8 submissions the program committee selected six papers. Each of these papers was reviewed by three or four program committee members. The workshop began with an invited talk by Sharad Malik on *Experiences with Quantified Boolean Formula Solvers*.

We thank the program committee for their effort in evaluating the articles and giving helpful comments to the authors. We also thank the organizers of the hosting conference, CAV'05, Kousha Etessami and Sriram Rajamani.

Organizers

Armin Biere
Johannes Kepler University, Linz, Austria

Ofer Strichman
Technion, Haifa, Israel

Program Comittee

Per Bjesse
Synopsys, USA

Alessandro Cimatti
IRST, Italy

Koen Claessen
Chalmers, Sweden

Ranan Fraer
Intel, Israel

Danny Geist
IBM Israel

Alan Hu
UBC, Canada

Yunshan Zhu
Synopsys, USA

João Marques Silva
Lisbon, Portugal

Ken McMillan
Cadence, USA

Fabio Somenzi
University of Colorado, USA

Armin Biere, Ofer Strichman

Linz, Haifa, June 2005

Contents

Preface 3

Contents 5

Invited Talk: Sharad Malik
Experiences with QBF Solvers 7

Jehle, Johannsen, Lange, Rachinsky
Bounded Model checking for all Regular Properties 9

Franzén
Using Satisfiability Modulo Theories for Inductive Verification of LUSTRE programs 23

Shacham, Yorav
Adaptive Application of SAT Solving Techniques 37

Awedh, Somenzi
Termination Criteria for Bounded Model Checking: Extensions and Comparison 51

Geist, Ginzburg, Lustig, Shacham, Rabinovitz, Tzoref
Supporting SAT based BMC on Finite Path Models 65

Kroening
Computing Over−Approximations with Bounded Model Checking 75

Authors 89

Invited Talk

Experiences with QBF Solvers

Sharad Malik
Princeton University

Abstract
Recent success with DPLL style search based SAT solvers has prompted efforts in extending these techniques to solvers for Quantified Boolean Formulas (QBFs). One of the motivations for developing QBF solvers is to tackle the problem of determining the diameter of the state space of sequential circuits. This is especially relevant for Bounded Model Checking (BMC), where this diameter provides a bound for the unrolling required to make BMC complete.

I will first describe extensions to DPLL search techniques developed for QBF solvers and experiences with using these solvers for the diameter problem. Next, I will consider a special case of QBF, 2QBF, which limits the depth of quantification to two and discuss specialized techniques for this restricted form. This is again motivated by the diameter problem for which this form suffices. Then, I will comment on the inherent complexity of QBF for tackling the diameter problem. Finally I will describe a current effort to consider alternatives to search based QBF solvers by considering the implementation of quantification operators directly on logic circuits.

Bounded Model Checking for All Regular Properties

Markus Jehle Jan Johannsen Martin Lange
Nicolas Rachinsky

Institut für Informatik, LMU München, Germany

Abstract

The technique of bounded model checking is extended to the linear time μ-calculus, a temporal logic that can express all monadic second-order properties of ω-words, in other words, all ω-regular languages. Experimental evidence is presented showing that the method can be successfully employed for properties that are hard or impossible to express in the weaker logic LTL that is traditionally used in bounded model checking.

Key words: model checking, satisfiability solving, expressiveness

1 Introduction

Bounded model checking is a verification technique for linear time properties. Only paths of a certain length through a transition system are considered. It is therefore not complete but only an approximation method relying on the fact that unsatisfied formulas often have short counterexamples.

On the other hand, the boundedness plus the fact that models are linear structures make the problem suitable for a reduction to SAT - the satisfiability problem for propositional logic. It is known from a different symbolic technique, namely BDD-based model checking [5], that transition systems can be encoded as boolean functions, and that these encodings can be significantly smaller than explicit representations.

So far, bounded model checking has been employed for LTL [10] and variants thereof. But the expressive power of LTL is rather limited: it is equiexpressive to First-Order Logic over ω-words,resp. star-free languages [14].

There are various temporal specification languages for ω-regular languages: ETL [18] and QPTL [13] extend the syntax of LTL with Büchi automata, resp. propositional quantifiers. The are not very usable because of an infinite set of temporal connectives, resp. complexity issues. Dynamic LTL [7] simply obtains ω-regular expressive power by adding ω-regular expressions to LTL; industrially used logics like FTL [1] and PSL/Sugar [3] are geared towards

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

usability and, thus, provide a very rich syntax; and the linear time μ-calculus μTL [2] simply achieves ω-regular power by replacing the *until* operator by a general-purpose least fixpoint quantifier.

Inspired by the success that bounded model checking for LTL has had so far [4], we show how to do bounded model checking for μTL. The choice of μTL is motivated in two ways. First, since it is a natural extension of LTL, there is reason to believe that many optimisations that have been found for bounded model checking LTL carry over to μTL. Second, just like the modal μ-calculus, it provides a framework which other specification formalisms can often easily be translated into. Hence, bounded model checking for μTL has the potential to implicitly provide bounded model checking procedures for other languages as well.

Unlike the modal μ-calculus, μTL does not have a strict alternation hierarchy. Therefore, every μTL formula can be transformed into an equivalent alternation-free formula. This translation is exponential in the alternation depth of the original formula. However, formulas with a lot of alternation are hardly seen as specifications because they are not easy to read. The encoding into SAT presented here makes use of this result.

The rest of the paper is organised as follows. Section 2 recalls μTL. Section 3 compares LTL and μTL using some example formulas. Section 4 defines a bounded semantics for μTL along the same lines as the one for LTL [4]. Section 5 contains the reduction from μTL formulas over paths of bounded length into SAT. Section 6 reports on a prototype implementation of this translation and presents experimental results.

What remains to do done is to check which known optimisations for LTL bounded model checking can be transferred to μTL, to also find small completeness thresholds like it was done for LTL [4,6], etc.

2 Preliminaries

2.1 The Linear Time μ-Calculus μTL

Let $\mathcal{P}$ be a set of propositions which contains $\mathtt{tt}$ and $\mathtt{ff}$ and is closed under complementation, i.e., for every $q \in \mathcal{P}$ there is an $\bar{q} \in \mathcal{P}$ with $\bar{\bar{q}} = q$. Let $\mathcal{V}$ be a set of monadic second-order variables. Formulas of μTL in positive normal form are given by the following grammar.

$$\varphi \quad ::= \quad q \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \mu X.\varphi \mid \nu X.\varphi$$

where $q \in \mathcal{P}$ and $X \in \mathcal{V}$. The set $Sub(\varphi)$ of subformulas of φ is defined as usual, e.g. $Sub(\mu X.\varphi) := \{\mu X.\varphi\} \cup Sub(\varphi)$.

Formulas are assumed to be well-named, i.e., no variable is bound more than once in a formula. Then for each $\varphi \in \mu$TL there is a function $fp_\varphi : \mathcal{V} \cap Sub(\varphi) \to Sub(\varphi)$ that maps each variable X occurring in φ to its defining fixpoint formula $\sigma X.\psi$. If $fp_\varphi(X)$ is $\mu X.\psi$ for some formula ψ, we say that X

is of type μ, otherwise X is of type ν.

A total, labeled transition system (LTS) is a tuple $\mathcal{T} = (\mathcal{S}, \longrightarrow, \mathcal{I}, S_0)$ where $\mathcal{S}$ is a set of states. $\longrightarrow$ is a binary relation on states s.t. for every $s \in \mathcal{S}$ there is a $t \in \mathcal{S}$ with $s \longrightarrow t$. $\mathcal{I} : \mathcal{P} \to 2^{\mathcal{S}}$ interprets the propositional constants from $\mathcal{P}$ in $\mathcal{T}$ respecting $\mathtt{tt}$, $\mathtt{ff}$ and complementation. $S_0 \subseteq \mathcal{S}$ is the set of all *starting* states.

A path through $\mathcal{T}$ is an infinite sequence $\pi = s_1 s_2 \ldots$, s.t. $s_1 \in S_0$ and for all $i \in \mathbb{N}$: $s_i \longrightarrow s_{i+1}$.

We write π^k for the k-th state of π, $Pos(\pi)$ for the set of states in π, and $Pos^k(\pi)$ for $\{\pi^i \in Pos(\pi) \mid i \leq k\}$.

Formulas of μTL are interpreted over a path $\pi = s_1 s_2 \ldots$ of an LTS $\mathcal{T}$. Free variables are interpreted using an *environment* $\rho : \mathcal{V} \to 2^{Pos(\pi)}$. With $\rho[X \mapsto T]$ we denote the function that maps X to T and behaves like ρ on all other arguments. Since π will always be derivable from the context we avoid mentioning it explicitly.

$$\begin{aligned}
[\![q]\!]_\rho &:= \mathcal{I}(q) \\
[\![X]\!]_\rho &:= \rho(X) \\
[\![\varphi \vee \psi]\!]_\rho &:= [\![\varphi]\!]_\rho \cup [\![\psi]\!]_\rho \\
[\![\varphi \wedge \psi]\!]_\rho &:= [\![\varphi]\!]_\rho \cap [\![\psi]\!]_\rho \\
[\![\bigcirc \varphi]\!]_\rho &:= \{\pi^k \mid \pi^{k+1} \in [\![\varphi]\!]_\rho\} \\
[\![\mu X.\varphi]\!]_\rho &:= \bigcap \{T \subseteq Pos(\pi) \mid [\![\varphi]\!]_{\rho[X \mapsto T]} \subseteq T\} \\
[\![\nu X.\varphi]\!]_\rho &:= \bigcup \{T \subseteq Pos(\pi) \mid T \subseteq [\![\varphi]\!]_{\rho[X \mapsto T]}\}
\end{aligned}$$

We write $\pi^k \models_\rho \varphi$ if $\pi^k \in [\![\varphi]\!]_\rho$. If φ is closed, i.e., it does not contain any free variables we write $\pi^k \models \varphi$ instead. Finally, we write $\pi \models \varphi$ if $\pi^1 \in [\![\varphi]\!]$.

Lemma 2.1 *For every closed $\varphi \in \mu$TL, there is a closed $\overline{\varphi} \in \mu$TL s.t. for all paths π of all LTSs $\mathcal{T}$: $\pi \models \varphi$ iff $\pi \not\models \overline{\varphi}$.*

Proof. The complement $\overline{\varphi}$ can inductively be constructed using complementation closure of atomic propositions, deMorgan's laws and the rules $\overline{\overline{\psi}} := \psi$, $\overline{\bigcirc \psi} := \bigcirc \overline{\psi}$, $\overline{\mu X.\psi(X)} := \nu X.\overline{\psi(\overline{X})}$, and $\overline{\nu X.\psi(X)} := \mu X.\overline{\psi(\overline{X})}$. □

We also allow ourselves to write $\neg\varphi$ instead of $\overline{\varphi}$.

Approximants of a formula $\sigma X.\varphi$ w.r.t. a linear time structure π and an environment $\rho : \mathcal{V} \to 2^{Pos(\pi)}$ are defined for every $i \in \mathbb{N}$ as usual:

$$X^0_\rho := \begin{cases} \emptyset & \text{for } \sigma = \mu \\ Pos(\pi) & \text{for } \sigma = \nu \end{cases}, \qquad X^{i+1}_\rho = [\![\varphi]\!]_{\rho[X \mapsto X^i_\rho]}$$

The following is a standard results about fixpoint logics. It follows immediately from the Knaster-Tarski Theorem and the fact that the semantics of a

formula with a free variable is a monotone function on the subset lattice of states on a path.

Lemma 2.2 *For all $\varphi \in \mu\mathrm{TL}$ and environment ρ we have:*

$$[\![\mu X.\varphi]\!]_\rho = \bigcup_{i\in\mathbb{N}} X^i_\rho \ , \qquad [\![\nu X.\varphi]\!]_\rho = \bigcap_{i\in\mathbb{N}} X^i_\rho$$

We say that X depends on Y in φ, written $Y \prec_\varphi X$, if Y is free in $fp_\varphi(X)$. We write $\leq_\varphi$ for the reflexive-transitive closure of $\prec_\varphi$. The alternation depth $ad(\varphi)$ of φ is n if there is a maximal chain $X_0 \leq_\varphi \ldots \leq_\varphi X_n$ with consecutive variables having different fixpoint types. Let $\mu\mathrm{TL}^k := \{\varphi \mid ad(\varphi) \leq k\}$.

Proposition 2.3 [17,8] *Every closed $\varphi \in \mu\mathrm{TL}$ is equivalent to a closed $\varphi' \in \mu\mathrm{TL}^0$ s.t. $|\varphi'| = O(|\varphi| \cdot 2^{4\cdot ad(\varphi)})$.*

3 μTL vs. LTL

Formulas of LTL are built from atomic propositions using the boolean operators $\wedge$, $\vee$ and $\neg$, as well as the temporal operators $\bigcirc$ (next) and $\mathtt{U}$ (until) with their usual semantics [10].

Proposition 3.1 *For every formula $\varphi \in \mathrm{LTL}$ there is an equivalent $\varphi' \in \mu\mathrm{TL}^0$ s.t. $|\varphi'| = O(|\varphi|)$.*

It follows that μTL model checking over labelled transition systems is also PSPACE-hard [11] where the size of the input is the number of states in explicit representation. In fact, it is also PSPACE-complete [16].

Proposition 3.2 [2] *A language is ω-regular iff it is μTL-definable.*

Together with Proposition 2.3 we obtain that $\mu\mathrm{TL}^0$ is already capable of defining all ω-regular properties.

In the following, we will give a few examples of properties that are either μTL- but not LTL-definable, or that can be written down more succinctly in μTL.

Example 1 "Formula ψ holds on every even state of a path" is not LTL-definable, but can be expressed in μTL as $\nu X.\psi \wedge \bigcirc\bigcirc X$.

Example 2 Suppose we have a set $Q = \{q_0, \ldots, q_{n-1}\}$ of atomic propositions and require them to occur repeatedly in this order. This can be done in μTL with the following formula of size linear in n.

$$\varphi \ := \ \nu X.q_0 \wedge \bigcirc(q_1 \wedge \bigcirc(q_2 \wedge \ldots \bigcirc(q_n \wedge \bigcirc X)\ldots))$$

The property is still star-free, hence, LTL definable. But note that propositions do not exclude each other. Thus, an equivalent LTL formula would have to assert the label of the next state in accordance with the labels of the last

n states – for every starting point in the order $q_0, \ldots, q_{n-1}$. Hence, its size would be quadratic in n.

Example 3 The next formula describes the capacity property of a bounded message buffer of size n. A word $w \in \{push, pop, nop\}^\omega$ satisfies β_n if for every prefix v of w, the difference between the numbers of occurrences of *push* and *pop* in v is between 0 and n. This is also a star-free property, but for growing n it occurs arbitrarily high in the dot-depth hierarchy of star-free languages [15], and thus it is notoriously hard to formalize in LTL. The formula β_n is φ_0, where φ_i is inductively defined as follows.

$$\varphi_0 := \nu X_0.(push \rightarrow \bigcirc \varphi_1) \wedge \neg pop \wedge (nop \rightarrow \bigcirc X_0)$$
$$\varphi_i := \nu X_i.(push \rightarrow \bigcirc \varphi_{i+1}) \wedge (pop \rightarrow \bigcirc X_{i-1}) \wedge (nop \rightarrow \bigcirc X_i) \quad \text{if } 1 \leq i < n$$
$$\varphi_n := \nu X_n.\neg push \wedge (pop \rightarrow \bigcirc X_{n-1}) \wedge (nop \rightarrow \bigcirc X_n)$$

The size of β_n is obviously linear in n, whereas only exponential size LTL formulas specifying this property are known [12].

4 A Bounded Semantics for μTL

Assume an LTS $\mathcal{T} = (\mathcal{S}, \longrightarrow, \mathcal{I}, S_0)$ to be fixed and of finite size. Every path through $\mathcal{T}$ starting with a state in S_0 induces a linear time structure π.

Definition 1 *A path π of $\mathcal{T}$ is called a (k, ℓ)-*loop *for $\ell \leq k \in \mathbb{N}$ if $\pi^{k+1+i} = \pi^{\ell+i}$ for all $i \in \mathbb{N}$.*

Note that if φ is satisfied by a path of a finite transition system ($|\mathcal{S}| < \infty$), then it is already satisfied by a path which is a (k, ℓ)-loop for some ℓ, k with $\ell \leq k$. This is a consequence of Proposition 3.2. Small upper bounds on k – so-called completeness thresholds – remain to be found.

Definition 2 Given a $k \in \mathbb{N}$, a path π of $\mathcal{T}$ and an environment $\rho : \mathcal{V} \rightarrow Pos(\pi)$, we define the k-bounded semantics $[\![\varphi]\!]^k_\rho$ by distinguishing two cases:

Case 1, π is a (k, ℓ)-loop for some $\ell \leq k$: Then the bounded semantics does not differ from the unbounded semantics of Section 2, i.e. we define

$$[\![\varphi]\!]^k_\rho \;:=\; [\![\varphi]\!]_\rho$$

Case 2, π is not a (k, ℓ)-loop for any $\ell \leq k$: Then we define

$$\begin{aligned}
[\![q]\!]^k_\rho &:= \mathcal{I}(q) \cap Pos^k(\pi) \\
[\![X]\!]^k_\rho &:= \rho(X) \cap Pos^k(\pi) \\
[\![\varphi \vee \psi]\!]^k_\rho &:= [\![\varphi]\!]^k_\rho \cup [\![\psi]\!]^k_\rho \\
[\![\varphi \wedge \psi]\!]^k_\rho &:= [\![\varphi]\!]^k_\rho \cap [\![\psi]\!]^k_\rho \\
[\![\bigcirc \varphi]\!]^k_\rho &:= \{\pi^i \mid i < k \text{ and } \pi^{i+1} \in [\![\varphi]\!]^k_\rho\}
\end{aligned}$$

$$\begin{aligned}\llbracket \mu X.\varphi \rrbracket^k_\rho &:= \bigcap \{T \subseteq Pos^k(\pi) \text{ and } \llbracket \varphi \rrbracket^k_{\rho[X \mapsto T]} \subseteq T\} \\ \llbracket \nu X.\varphi \rrbracket^k_\rho &:= \emptyset\end{aligned}$$

As for the unbounded case, we define *bounded approximants* for the iterative evaluation of the bounded semantics of fixpoint formulas.

Definition 3 Bounded approximants *for least fixpoint formulas* $\mu X.\varphi$*, a* $k \in \mathbb{N}$*, a path* π *and an environment* ρ *are defined for all* $i \in \mathbb{N}$ *as*

$$X^{k,0}_\rho := \emptyset \ , \quad X^{k,i+1}_\rho := \llbracket \varphi \rrbracket^k_{\rho[X \mapsto X^{k,i}_\rho]}$$

For greatest fixpoint formulas, bounded approximants depend on the type of the underlying path. If π *is a* (k,ℓ)*-loop for some* $\ell \leq k$ *then we define*

$$X^{k,0}_\rho := Pos^k(\pi) \ , \quad X^{k,i+1}_\rho := \llbracket \varphi \rrbracket^k_{\rho[X \mapsto X^{k,i}_\rho]}$$

Otherwise, we set $X^{k,i}_\rho := \emptyset$ *for all* $i \in \mathbb{N}$*.*

The following lemmas form the basis for the correctness of the reduction in the next section. Lemma 4.1 expresses the monotonicity of the bounded semantics, and Lemma 4.2 states that the bounded approximants really approximate the bounded semantics. They are proved by simultaneous induction on the structure of μTL formulas, in a way similar to the corresponding statements for the unbounded semantics.

Lemma 4.1 *For all* $k \in \mathbb{N}$*, all* $X \in \mathcal{V}$*, all* $\varphi \in \mu\text{TL}$*, all paths* π*, all environments* ρ *and all* $P \subseteq Q \subseteq Pos^k(\pi)$ *we have:* $\llbracket \varphi \rrbracket^k_{\rho[X \mapsto P]} \subseteq \llbracket \varphi \rrbracket^k_{\rho[X \mapsto Q]}$*.*

Lemma 4.2 *For all* $k \in \mathbb{N}$*, all* $X \in \mathcal{V}$*, all environments* ρ*, all* $\varphi \in \mu\text{TL}$ *and all paths* π *we have:* $\llbracket \mu X.\varphi \rrbracket^k_\rho = \bigcup_{i \in \mathbb{N}} X^{k,i}_\rho$ *and* $\llbracket \nu X.\varphi \rrbracket^k_\rho = \bigcap_{i \in \mathbb{N}} X^{k,i}_\rho$*.*

The following lemma states that the bounded semantics is an under-approximation of the unbounded semantics. This entails that any counterexample found by bounded model checking is an actual counterexample to the checked specification.

Lemma 4.3 *For all* $\varphi \in \mu\text{TL}$*, all environments* ρ*, all* $k \in \mathbb{N}$ *and all paths* π *we have:* $\llbracket \varphi \rrbracket^k_\rho \subseteq \llbracket \varphi \rrbracket_\rho$*.*

Proof. The only interesting case is the one of φ being $\mu X.\psi$, and the path π is not a (k,ℓ)-loop for any ℓ. For this case, we prove by a side induction on i that $X^{k,i}_\rho \subseteq X^i_\rho$ for all $i \in \mathbb{N}$, from which the lemma follows by Lemmas 4.2 and 2.2. The induction basis for the claim is trivial. For the induction step, note that

$$X^{k,i+1}_\rho = \llbracket \psi \rrbracket^k_{\rho[X \mapsto X^{k,i}]} \subseteq \llbracket \psi \rrbracket_{\rho[X \mapsto X^{k,i}_\rho]} \subseteq \llbracket \psi \rrbracket_{\rho[X \mapsto X^i_\rho]} = X^{i+1}_\rho$$

where the first inclusion follows by the main induction hypothesis, and the second one by the side induction hypothesis and monotonicity. □

The next lemma shows that the bounded semantics is monotone in the bound k. This entails that by increasing the bound, one does not lose any counterexamples that would have been found with a smaller bound.

Lemma 4.4 *For all $k \in \mathbb{N}$, all $\varphi \in \mu\mathrm{TL}$, all environments ρ and all paths π we have: $[\![\varphi]\!]^k_\rho \subseteq [\![\varphi]\!]^{k+1}_\rho$.*

Proof. The only non-trivial case is the one of π not being a $(k+1,\ell)$-loop for any $\ell \leq k+1$. Again, the proof is by induction on φ. The only interesting case is $\varphi = \mu X.\psi$, where we prove by side induction on i that $X^{k,i}_\rho \subseteq X^{k+1,i}_\rho$, from which the claim follows by Lemma 4.2. For $i = 0$ this is trivial again, and the inductive step follows by

$$X^{k,i+1}_\rho \;=\; [\![\psi]\!]^k_{\rho[X\mapsto X^{k,i}_\rho]} \;\subseteq\; [\![\psi]\!]^{k+1}_{\rho[X\mapsto X^{k,i}_\rho]} \;\subseteq\; [\![\psi]\!]^{k+1}_{\rho[X\mapsto X^{k+1,i}_\rho]} \;=\; X^{k+1,i+1}_\rho$$

where the first inclusion follows by the main induction hypothesis, and the second one by the side induction hypothesis and Lemma 4.1. □

Lemma 4.5 *For any $\sigma \in \{\mu,\nu\}$, any formula φ, environment ρ, and $k \in \mathbb{N}$ we have $[\![\sigma X.\varphi]\!]^k_\rho = X^{k,k}_\rho$.*

Proof. This is a consequence of Lemma 4.2, since the chain of bounded approximants must become stationary after at most k steps. The reason is that all bounded approximants are subsets of $Pos^k(\pi)$, and $|Pos^k(\pi)| = k$. □

By use of this lemma, for a fixpoint formula φ containing m nested fixpoint operators, $[\![\varphi]\!]^k$ can be computed in k^m steps. For alternation-free formulas in $\mu\mathrm{TL}^0$ one can do better. We present the construction for least fixpoints, for greatest fixpoints it is completely analogous.

Let $\varphi = \mu X.\psi$ be a closed fixpoint formula, and let $X = X_1,\ldots,X_r$ be those variables in φ that depend on X, i.e., $X \leq_\varphi X_i$ for $i = 1,\ldots,r$. Since $\varphi \in \mu\mathrm{TL}^0$, all the variables X_i are of type μ. Now φ is transformed into a system of equations

$$\begin{array}{rcl} X_1 & = & \psi_1(X_1,\ldots,X_r) \\ & \vdots & \\ X_r & = & \psi_r(X_1,\ldots,X_r) \end{array} \qquad (1)$$

where the formulas ψ_j contain no fixpoint subformulas that depend on the variables $X_1,\ldots,X_r$, i.e., every fixpoint subformula of $\psi_j(X_1,\ldots,X_r)$ is a subformula of some closed fixpoint subformula of $\psi_j(X_1,\ldots,X_r)$. The translation is obtained as follows: let

$$fp_\varphi(X_i) \;=\; \mu X_i.\psi_i(X_1,\ldots,X_i,\mu Y_1.\theta_1,\ldots,\mu Y_s.\theta_s)$$

containing free variables among $X_1,\ldots,X_{i-1}$, where the subformulas $\mu Y_j.\theta_j$ for Y_j among $X_{i+1},\ldots,X_r$ are those outermost fixpoint subformulas of ψ_i that

contain any free variables from $X_1, \ldots, X_i$. This formula yields the equation $X_i = \psi_i(X_1, \ldots, X_i, Y_1, \ldots, Y_s)$ in (1).

For the system of equations (1), the bounded simultaneous approximants $X_i^{k,(j)}$ for $1 \leq i \leq r$ and $j \in \mathbb{N}$ are inductively defined as follows:

$$X_i^{k,(0)} = \emptyset \qquad X_i^{k,(j+1)} = [\![\psi_i(X_1, \ldots, X_r)]\!]_{\rho_j}^k \tag{2}$$

where ρ_j is the environment that maps each variable X_h to $X_h^{k,(j)}$, for $1 \leq h \leq r$.

Lemma 4.6 *For a closed fixpoint formula $\mu X.\varphi$ as above, $[\![\mu X.\varphi]\!]^k = X_1^{k,(kr)}$.*

Proof. The fixpoint of the simultaneous iteration (2) is the same as $[\![\mu X.\varphi]\!]^k$ by Békič' Theorem. Moreover, (2) reaches its fixpoint after at most $k \cdot r$ iterations, since there are r subsets of $Pos^k(\pi)$ being computed, and in the worst case, in each iteration only one of the sets increases by one element. □

5 The Reduction to SAT

5.1 Symbolic Representations

Propositional Logic over a set $\mathcal{V}$ of propositional variables is the closure of $\mathcal{V}$ under the boolean connectives $\neg$, $\vee$, and consequently also $\wedge$, $\rightarrow$, etc. Here we assume a finite LTS $\mathcal{T} = (\mathcal{S}, \longrightarrow, \mathcal{I}, S_0)$ to be given symbolically, i.e., by propositional formulas

- $f_{\text{start}} : \mathbb{B}^n \rightarrow \mathbb{B}$ with $f_{\text{start}}(\bar{x}) = \mathtt{tt}$ iff $\bar{x} \in S_0$,
- $f_q : \mathbb{B}^n \rightarrow \mathbb{B}$ for every $q \in \mathcal{P}$ with $f_q(\bar{x}) = \mathtt{tt}$ iff $\bar{x} \in \mathcal{I}(q)$,
- $f_{\text{trans}} : \mathbb{B}^{2 \cdot n} \rightarrow \mathbb{B}$ with $f_{\text{trans}}(\bar{x}, \bar{y}) = \mathtt{tt}$ iff $\bar{x} \longrightarrow \bar{y}$.

where $n := \lceil \log |\mathcal{S}| \rceil$. I.e. every state is identified by a unique number in binary coding.

Most SAT solvers expect that the input formula is given in conjunctive normal form (CNF). Our translation as defined below produces arbitrary formulas, but it is well-known that such formulas can be translated into CNF with only a linear blow-up in size and a linear number of additional variables.

5.2 The Translation

For a symbolically represented transition system $\mathcal{T}$ with 2^n states, a formula $\varphi \in \mu\text{TL}^0$ and a $k \in \mathbb{N}$ we define a boolean formula $\langle\!\langle \mathcal{T}, \varphi \rangle\!\rangle^k$ in the following variables:

- the *path variables* $\bar{s}_i = s_{i,1}, \ldots, s_{i,n}$ for $1 \leq i \leq k$, coding the i-th state on a path.
- auxiliary variables $v(X)_i$ for every second-order variable X and $1 \leq i \leq k$. These variables will not occur in the final formula $\langle\!\langle \mathcal{T}, \varphi \rangle\!\rangle^k$, they are only

used during the construction as placeholders for free variables in subformulas.

- the *approximant variables* $a(X,j)_i^k$ and $a(X,j)_i^{k,\ell}$ for every second-order variable X and $1 \leq i, \ell \leq k$ and $j \in \mathbb{N}$. These variables express that state i is in the bounded approximant $X^{k,(j)}$.

First, we define a formula $\langle\!\langle\, \mathcal{T} \,\rangle\!\rangle^k$ saying that the path variables $\bar{s}_1, \ldots, \bar{s}_k$ actually encode a path in $\mathcal{T}$ by

$$\langle\!\langle\, \mathcal{T} \,\rangle\!\rangle^k \;:=\; f_{\text{start}}(\bar{s}_1) \wedge \bigwedge_{i=1}^{k-1} f_{\text{trans}}(\bar{s}_i, \bar{s}_{i+1}) \;.$$

Next, as usual we define formulas to distinguish between the cases where the path is a (k,ℓ)-loop for $\ell \leq k$, and where it is not, by

$$\text{Loop}^{k,\ell} \;:=\; f_{\text{trans}}(\bar{s}_k, \bar{s}_\ell) \qquad \neg\text{Loop}^k \;:=\; \bigwedge_{i=1}^{k} \neg\text{Loop}^{k,i}$$

and using these, we define the translation by

$$\langle\!\langle\, \mathcal{T}, \varphi \,\rangle\!\rangle^k \;:=\; \langle\!\langle\, \mathcal{T} \,\rangle\!\rangle^k \wedge \Big(\big(\neg\text{Loop}^k \wedge \langle\!\langle\, \varphi \,\rangle\!\rangle^k\big) \vee \bigvee_{\ell=1}^{k} \big(\text{Loop}^{k,\ell} \wedge \langle\!\langle\, \varphi \,\rangle\!\rangle^{k,\ell}\big) \Big) \;.$$

The formula $\langle\!\langle\, \varphi \,\rangle\!\rangle^k$ that actually encodes φ in the case of a non-loop is defined as $\langle\!\langle\, \varphi \,\rangle\!\rangle_1^k \wedge \mathit{Defs}(\varphi)^k$, where the formulas $\langle\!\langle\, \psi \,\rangle\!\rangle_i^k$ for subformulas ψ of φ and $1 \leq i \leq k$ express that the i^{th} state satisfies ψ. For formulas without fixpoint operators, these are inductively defined by:

$$\begin{aligned}
\langle\!\langle\, q \,\rangle\!\rangle_i^k &:= f_q(\bar{s}_i) \\
\langle\!\langle\, X \,\rangle\!\rangle_i^k &:= v(X)_i \\
\langle\!\langle\, \varphi \vee \psi \,\rangle\!\rangle_i^k &:= \langle\!\langle\, \varphi \,\rangle\!\rangle_i^k \vee \langle\!\langle\, \psi \,\rangle\!\rangle_i^k \\
\langle\!\langle\, \varphi \wedge \psi \,\rangle\!\rangle_i^k &:= \langle\!\langle\, \varphi \,\rangle\!\rangle_i^k \wedge \langle\!\langle\, \psi \,\rangle\!\rangle_i^k \\
\langle\!\langle\, \bigcirc \varphi \,\rangle\!\rangle_i^k &:= \begin{cases} \langle\!\langle\, \varphi \,\rangle\!\rangle_{i+1}^k & \text{if } i < k \\ \mathtt{ff} & \text{otherwise} \end{cases}
\end{aligned}$$

Next, we define the translation for a closed greatest fixpoint formula as the constant $\mathtt{ff}$,

$$\langle\!\langle\, \nu X.\psi \,\rangle\!\rangle_i^k \;:=\; \mathtt{ff} \;,$$

and for a closed least fixpoint formula as the approximant variable

$$\langle\!\langle\, \mu X.\psi \,\rangle\!\rangle_i^k \;:=\; a(X, kr)_i^k \;,$$

where r is the number of second-order variables Y in $\mu X.\psi$ with $X \leq_\varphi Y$.

Note that in a fixpoint formula, the bound variable can occur several times. Therefore a straightforward translation of the approximants by syntactic unfolding would lead to an exponential blowup. To prevent this, we use the approximant variables to abbreviate the approximants, and the formula $\mathit{Defs}(\varphi)^k$ takes care of their proper interpretation. It is defined as the conjunction of the defining formulas $\mathit{Def}(\psi)^k$, over all subformulas ψ of φ that are closed least fixpoint formulas.

Another exponential blowup would occur if nested fixpoints were translated straightforwardly inside out, since the unfolding of a formula with m nested fixpoints would produce k^m subformulas. Therefore we use the transformation of a closed least fixpoint subformula ψ into a system of r equations (1), as described at the end of Section 4:

$$\begin{aligned} X_1 &= \psi_1(X_1, \ldots, X_r) \\ &\vdots \\ X_r &= \psi_r(X_1, \ldots, X_r) \end{aligned}$$

The formula $\mathit{Def}(\psi)^k$ describes the evaluation of this system of equations by the simultaneous approximants (2) by giving definitions for the corresponding approximant variables. I.e., $\mathit{Def}(\psi)^k$ is the conjunction of the equivalences [1] $a(X,s)_i^k \leftrightarrow F(X_j,s)_i^k$ over all $1 \le j \le r$, $1 \le s \le kr$ and $1 \le i \le k$, where

- $F(X_j,1)_i^k$ is the translation $\langle\!\langle\, \psi_j(X_1,\ldots,X_r) \,\rangle\!\rangle_i^k$ with the variables $v(X_h)_g^k$ for $1 \le h \le r$ and $1 \le g \le k$ replaced by $\mathtt{ff}$, and
- $F(X_j,s)_i^k$ for $s > 1$ is $\langle\!\langle\, \psi_j(X_1,\ldots,X_r) \,\rangle\!\rangle_i^k$ with the variables $v(X_h)_g^k$ replaced by $a(X_h,s-1)_g^k$, for $1 \le h \le r$ and $1 \le g \le k$.

Similarly, the translation $\langle\!\langle\, \varphi \,\rangle\!\rangle^{k,\ell}$ of φ in the case of a loop is defined as $\langle\!\langle\, \varphi \,\rangle\!\rangle_1^{k,\ell} \wedge \mathit{Defs}(\varphi)^{k,\ell}$, where the inductive definition of the formulas $\langle\!\langle\, \psi \,\rangle\!\rangle_i^{k,\ell}$ differs only in the clause for $\bigcirc\psi$, which becomes:

$$\langle\!\langle\, \bigcirc\varphi \,\rangle\!\rangle_i^{k,\ell} \;:=\; \begin{cases} \langle\!\langle\, \varphi \,\rangle\!\rangle_{i+1}^{k,\ell} & \text{if } i < k \\ \langle\!\langle\, \varphi \,\rangle\!\rangle_{\ell}^{k,\ell} & \text{otherwise} \end{cases}$$

For both closed least and greatest fixpoint formulas we now define the translation by

$$\langle\!\langle\, \sigma X.\psi \,\rangle\!\rangle_i^{k,\ell} \;:=\; a(X,kr)_i^{k,\ell}\,,$$

where like above, r is the number of second-order variables Y in $\sigma X.\psi$ with $X \le_\varphi Y$.

The formula $\mathit{Defs}(\varphi)^{k,\ell}$ is the conjunction of the formulas $\mathit{Def}(\psi)^{k,\ell}$ over all closed least and greatest fixpoint subformulas of φ. For such a subformula, written as an equation system in the variables $X_1,\ldots,X_r$, the for-

[1] If the formulas are transformed into CNF, these equivalences need not be written, but are implicitly produced by the transformation. One only needs to identify the variable $a(X,s)_i^k$ with the new variable abbreviating the formula $F(X_j,s)_i^k$.

mula $Def(\psi)^{k,\ell}$ is defined exactly as $Def(\psi)^k$ above, only that for a variable of type ν, the defining formulas for the first approximant variables become $a(X_j,1)_i^{k,\ell} \leftrightarrow F(X_j,1)_i^{k,\ell}$, where in this case $F(X_j,1)_i^{k,\ell}$ is the translation $\langle\!\langle\, \psi_j(X_1,\ldots,X_r)\,\rangle\!\rangle_i^{k,\ell}$ with the variables $v(X_h)_g^{k,\ell}$ for $1 \le h \le r$ and $1 \le g \le k$ replaced by $\mathtt{tt}$.

The number of variables in and the size of the translation is measured in the numbers n, k, the size of the input formula s and the number of second-order variables v. They are easily estimated, and are – in the worst-case – as follows:

Proposition 5.1 *The formula $\langle\!\langle\, \mathcal{T}, \varphi \,\rangle\!\rangle^k$ contains $O(v^2k^3+kn)$ variables, and is of size $O(v^2k^3sn)$.*

Even though the number of variables produced by our translation is rather large, in particular regarding the cubic dependence on k, this might not be too problematic, since the approximant variables occur in $k+1$ disjoint parts of the formulas, each containing only $O(k^2)$ of them. Furthermore, note that it is only cubic for μTL formulas with multiple occurrences of variables under the scopes of different numbers of $\bigcirc$-operators. Hence, for LTL formulas the translation produces at most a quadratic number of variables.

Finally, we can easily observe the correctness of our translation, which is obvious from the definition for all cases except for the fixpoint formulas. But for those the correctness follows from Lemma 4.6.

Proposition 5.2 *The formula $\langle\!\langle\, \mathcal{T}, \varphi \,\rangle\!\rangle^k$ is satisfiable iff there is a path π in $\mathcal{T}$ starting at an initial state, and for which $\pi^0 \in [\![\varphi]\!]^k$.*

6 Experimental Results

The algorithm presented here is part of the verification tool μ-Sabre that is being developed at LMU Munich. The program is implemented in the lazy functional language Haskell using the Glasgow Haskell Compiler 6.2.2, with the exception of a small part of the program, dealing with linking of the SAT solver, that was implemented in C. The SAT solver used is version 2004.5.13 of zChaff [9].

The tests were carried out on a machine with two Intel® Xeon™ 2.4 GHz processors and 4GB of RAM. The second processor remained unused.

In a first test series we consider the property "there is a path with a b at an even position and a c at an odd position" on a family $\{\mathcal{T}_n \mid n \in \mathbb{N}\}$ of transition systems, s.t. $\mathcal{T}_n$ has got n states. The transitions between these states and their labels are as follows.

$$a \qquad a \qquad a \qquad\qquad a \qquad b \qquad c$$

The only starting state is the leftmost. The property is written in μTL as

n	Var	Cls	Red	SAT
22	6 k	42 k	0.24	0.09
32	13 k	97 k	0.86	1.87
42	23 k	191 k	2.88	4.49
52	36 k	298 k	6.29	26.83
62	52 k	435 k	12.13	3.00
72	71 k	647 k	21.10	21.01
82	92 k	847 k	35.09	107.17
92	116 k	1059 k	54.94	138.97

n	Var	Cls	Red	SAT
102	143 k	1322 k	91.29	22.05
112	173 k	1597 k	124.44	213.27
122	207 k	1915 k	178.86	462.75
132	242 k	2438 k	253.42	421.27
142	280 k	2831 k	338.51	1167.54
152	320 k	3229 k	469.33	630.07
162	366 k	3699 k	583.69	10.78
172	409 k	4128 k	805.48	865.44

Fig. 1. The even b / odd c example.

$(\mu X.b \vee \bigcirc\bigcirc X) \wedge (\mu Y.\bigcirc c \vee \bigcirc\bigcirc Y)$. It may not be an interesting property but we include it here because it cannot be formalised in LTL, c.f. Example 1.

The running times of our reduction (Red) and the SAT solver (SAT) are presented in Figure 1. The time unit is seconds. We only present satisfiable instances, i.e. those of even n. The table also contains the number of propositional variables (Var) and the number of clauses (Cls) in the resulting formulas – truncated down to multiples of 1000 in order to save space.

Our other tests use a transition system $\mathcal{B}_n$ modeling a message buffer of size n, holding messages that are single bits. Every state in $\mathcal{B}_n$ has $2n+3$ bits: The first two are the opcode for the next operation. The third bit is the output of the previous operation; its value is only specified in states following a *pop* operation. The remaining $2n$ bits represent the n buffer cells, each cell being represented by one bit indicating whether the cell is occupied, and the other being the value stored in the cell. The value of the second bit is unspecified for unoccupied cells.

The boolean formulas f_{start} and f_{trans} are hand-coded, with f_{start} saying that the buffer is initially empty, and f_{trans} specifying the changes in the buffer depending on the opcode, e.g., one disjunct of $f_{\text{trans}}(x, y)$ is

$$\neg x_1 \wedge \neg x_2 \wedge \bigwedge_{4 \le i \le 2n+3} (x_i \leftrightarrow y_i)$$

stating that a *nop* (having opcode 00) does not change the buffer content.

We test the property $\neg\beta_{n-1}$ of Example 3 on $\mathcal{B}_n$ in order to have a satisfiable example. The minimal counterexample showing that β_{n-1} is violated is a sequence of n *push* operations, thus in our second experiment we test whether $\mathcal{B}_n \models_n \neg\beta_n$, for various n. The results are shown in Figure 2. Again, the time unit is seconds.

In the third experiment, in order to see the dependence of the performance on the bound k, we test $\mathcal{B}_n \models_k \neg\beta_{n-1}$ for various values of $k \ge n$, for fixed $n = 12$. The results are presented in Figure 3.

The example formula β_n was chosen for two reasons: First, as mentioned above, the property expressed can probably not easily and succinctly be stated

n	Var	Cls	Red	SAT
6	15 k	55 k	0.35	0.24
7	28 k	98 k	0.75	3.47
8	48 k	163 k	1.68	2.67
9	75 k	256 k	3.56	4.71
10	114 k	384 k	7.31	11.18
11	165 k	554 k	14.36	13.34
12	231 k	775 k	27.20	27.16
13	316 k	1056 k	49.56	39.64

n	Var	Cls	Red	SAT
14	423 k	1407 k	89.46	56.11
15	554 k	1840 k	158.40	95.49
16	713 k	2364 k	253.85	188.07
17	905 k	2994 k	392.49	146.14
18	1133 k	3741 k	608.33	157.17
19	1401 k	4620 k	947.79	293.46
20	1715 k	5646 k	1362.74	226.30
21	2078 k	6833 k	2072.00	810.71

Fig. 2. The buffer example with $k = n$

k	Var	Cls	Red	SAT
12	231 k	775 k	27.09	27.02
14	309 k	1030 k	49.92	41.57
16	398 k	1321 k	86.66	42.10
18	498 k	1648 k	145.22	73.22
20	610 k	2011 k	218.77	66.15
22	732 k	2409 k	308.51	178.23
24	866 k	2843 k	425.32	380.42

k	Var	Cls	Red	SAT
26	1010 k	3312 k	590.76	34.09
28	1166 k	3818 k	780.84	182.86
30	1333 k	4359 k	1036.56	233.37
32	1511 k	4935 k	1317.30	216.08
34	1701 k	5548 k	1659.06	161.38
36	1901 k	6196 k	2171.02	718.25
38	2113 k	6880 k	2929.20	409.74

Fig. 3. The buffer example with $n = 12$.

in LTL. Second, it fully utilizes the syntactic possibilities of alternation-free μTL, since β_n has n nested fixpoints, and, due to the presence of the *nop* operation, each bound variable (except for X_n) occurs twice.

References

[1] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property specification language. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'02*, volume 2280 of *LNCS*, pages 296–311, Grenoble, France, 2002. Springer.

[2] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Conf. Record of the 13th Annual ACM Symp. on Principles of Programming Languages, POPL'86*, pages 173–183. ACM, 1986.

[3] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Proc. 13th Int. Conf. on Computer Aided Verification, CAV'01*, volume 2102 of *LNCS*, pages 363–367, 2001.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Proc. 5th Int. Conf. on Tools and Algorithms for the Analysis and Construction of Systems, TACAS'99*, volume 1579 of *LNCS*, Amsterdam, NL, Mar. 1999.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[6] E. Clarke, D. Kroening, O. Strichman, and J. Ouaknine. Completeness and complexity of bounded model checking. In *Proc. 5th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI'04*, volume 2937 of *LNCS*, pages 85–96. Springer, 2004.

[7] J. G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.

[8] M. Lange. Weak automata for the linear time μ-calculus. In R. Cousot, editor, *Proc. 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 267–281, 2005.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pages 530–535, 2001.

[10] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. on Foundations of Computer Science, FOCS'77*, pages 46–57, Providence, RI, USA, Oct. 1977. IEEE.

[11] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

[12] A. P. Sistla, E. M. Clarke, N. Francez, and A. R. Meyer. Can message buffers be axiomatized in linear temporal logic? *Information and Control*, 63(1-2):88–112, 1984.

[13] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1987.

[14] W. Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42(2):148–156, Aug. 1979.

[15] W. Thomas. A concatenation game and the dot-depth hierarchy. In E. Börger, editor, *Computation Theory and Logic*, volume 270 of *Lecture Notes in Computer Science*, pages 415–426. Springer, 1987.

[16] M. Y. Vardi. A temporal fixpoint calculus. In ACM, editor, *Proc. Conf. on Principles of Programming Languages, POPL'88*, pages 250–259, NY, USA, 1988. ACM Press.

[17] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, Nov. 1994.

[18] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

Using Satisfiability Modulo Theories for Inductive Verification of Lustre Programs

Anders Franzén [1]

Dip. Informatica e Telecomunicazioni
Università di Trento
Trento, Italy

Abstract

The problem of verifying safety properties of Lustre programs with integer arithmetic have been attacked in several different ways. Abstract interpretation is used in NBAC, and inductive verification using a SAT solver is used in Luke.

This paper presents a method of using Satisfiability Modulo Theories (SMT) as an incremental decision procedure for inductive verification of Lustre program. We show that even a very naive approach using SMT is competitive and in some instances complementary to other approaches.

Key words: Lustre, formal verification, temporal induction, satisfiability modulo theories

1 Introduction

In recent years, a new type of decision procedure for decidable fragments of first order logics have been developed in the field often called Satisfiability Modulo Theories (SMT). SMT can be seen as an extension of satisfiability for propositional logic with constraints in a specific theory, for instance linear arithmetic over the integers or reals. A common approach for decision procedures for these logics is to extend an ordinary SAT solver with the capability to handle constraints in the theory. This is the approach used in systems like CVCLite[1], DPLL(T)[12], haRVey[7] and MathSAT[3]. Some systems, like ARIO[19] are using a Pseudo-Boolean SAT solver instead, but is still using the same principles.

Here we will show how an incremental SMT procedure can be constructed with simple means, and how this procedure can be used for inductive verification of Lustre[15] programs. The resulting program will be demonstrated

[1] Email: anders@dit.unitn.it

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to be in many cases comparable in performance to tools using other methods, and in some cases superior.

1.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of checking satisfiability of quantifier-free first order formulas with respect to a particular theory. The theory of interest here is quantifier-free relations over linear arithmetic expressions, where all variables are constrained to be integer. Without loss of generality, we can restrict ourselves to relations of these types:

$$\sum_i a_i x_i = b$$
$$\sum_i a_i x_i \leq b$$

where the x_is are free variables, and a_i and b are (integer) constants. Relations like these will be called *constraints* in this paper. We can then create formulas like these:

$$(x \leq 0 \vee y \leq 0) \wedge -y \leq -1$$
$$x = 0 \wedge (P \rightarrow x \geq 0) \wedge (\neg P \rightarrow x \leq -1)$$

where x and y are integer variables and P is a propositional variable (0-ary predicate). Deciding satisfiability for formulas is done in a iterative algorithm. A SAT *enumerator* is used, which interprets the formula as purely propositional, and enumerates all satisfying truth assignments to propositional variables and constraints. These are then checked for satisfiability in the theory of integer linear arithmetic.

As a small example take the formula

$$x = 0 \wedge (P \rightarrow x \geq 0) \wedge (\neg P \rightarrow x \leq -1)$$

The SAT enumerator may start with the model $\{x = 0, \neg P, x \geq 0, x \leq -1\}$. Checking this model for satisfiability in the theory of integer linear arithmetic means to check if the system

$$x = 0$$
$$x \geq 0$$
$$x \leq -1$$

has a solution. It has none, so the SAT enumerator creates a new propositional model, say $\{x = 0, P, x \geq 0, \neg x \leq -1\}$. Now the system

$$x = 0$$
$$x \geq 0$$
$$x \geq 0$$

have to be checked, and since it is satisfiable, the formula is satisfiable, and we can create a model for it by merging a solution to the above constraint

problem with the propositional model from the SAT enumerator. The SAT enumerator is usually implemented as a SAT solver, and requesting a new model is done by adding a "conflict clause" describing the reason for unsatisfiability to the propositional formula held by the SAT solver, prohibiting the same propositional model from recurring.

1.2 The Lustre Programming Language

Lustre is a synchronous declarative language, whose basic data abstraction is the stream. A Lustre program is a *node*, with zero or more input streams and one or more output streams. A stream is a sequence of values, and Lustre supports boolean, integer and real streams. A node can also have internal streams. As a short example, consider this Lustre node.

```
node AddIf( X : int, B :bool ) returns ( Y : int );
let
    Y = if B then X+1 else X;
tel
```

The node has two input streams X, and B and one output stream Y. Given the input streams $\mathsf{X} = \{1, 2, 3, 4, 5, \ldots\}$ and $\mathsf{B} = \{$ false, true, false, true, false, $\ldots\}$ we would get the output stream $\mathsf{Y} = \{1, 3, 3, 5, 5, \ldots\}$. More information on the Lustre language can be found in [15,14].

2 The Basic Procedure

The verification method we are going to use is *temporal induction* [18,2], which is simply induction with depth over time, with one small addition to make it complete for finite state systems. The induction proof consists of two parts; the base case where we prove that the property hold in the k first states, and the step case where we prove that if it holds in k successive states, it also holds in the next state. The necessary modification from ordinary induction is that we need to require in the step case that the states are unique.

Here we will use this procedure to verify safety properties for Lustre programs with unbounded integers. This creates a infinite state system for which induction is not complete. In the case of Lustre, this is not such bad news as it may seem. In fact, Lustre with unbounded integers is Turing-complete[11]. This means that there exists no complete methods for verification of Lustre programs with unbounded integers.

There are several possible ways of doing induction, the induction scheme we use here is ZigZag[10].

In order to support unbounded integers, a decision procedure for propositional logic extended with linear constraints over the integers will be used.

2.0.1 Translation of Lustre Expressions

Boolean streams can be translated in a straightforward way to the logic description of the transition function. For instance the logical or operator $p = a$ **or** b can be translated to

$$(\neg a \vee p) \wedge (\neg b \vee p) \wedge (a \vee b \vee p)$$

Translating integer stream requires a little more care. To see why, let's look at the translation of integer if-then-else expressions in Lustre. An if-then-else expression looks like this

$$e = \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2$$

These could be translated in a straightforward way to

$$b \rightarrow e = e_1$$

$$\neg b \rightarrow e = e_2$$

but this is inefficient since when the SAT solver assigns a value to b only one of the expressions e_1 and e_2 is relevant, and this means that only one of the equalities should be checked for satisfiability. The solution used here to use guarded constraints $g \rightarrow c$ where g is a propositional variable called a *guard* and c is a constraint. This will make it possible to disregard constraints, since when the SAT solver assigns the guard to false, the corresponding constraint is irrelevant to the satisfiability of the formula. The formula for an if-then-else can then (simplified) become

$$b \rightarrow g_1 \wedge \neg g_2$$

$$\neg b \rightarrow \neg g_1 \wedge g_2$$

$$g_1 \rightarrow e = e_1$$

$$g_2 \rightarrow e = e_2$$

In general, this does not suffice to limit the size of the constraint problems, to those which are relevant with respect to the truth assignment of conditions in if-the-else expressions. Since e_1 and e_2 above can contain arbitrary integer expressions including if-then-else expressions, we would like to be able to disregard all constraints that occur in those expressions. A description of how this is accomplished can be found in [11].

The translation starts with the parse tree, where the property (a boolean Lustre stream) is at the root. The parse tree is traversed depth first, translating each node in turn. The clauses with the guarded constraints $g \rightarrow c$ are never created. Instead, a mapping between the guards and their respective constraints is maintained in the solver.

For every propositional model constructed by the SAT solver the truth assignment of the guards is checked. A constraint problem is generated from the constraints whose guards are true in the model. If the constraint problem is satisfiable, a model can be constructed by combining the propositional

model with the model to the constraint problem. Otherwise a *reason* for the unsatisfiability is extracted from the constraint problem.

A reason is a subset of constraints which causes the unsatisfiability. A clause comprising the negation of the guards for the constraints in the reason is added to the SAT formula, prohibiting the SAT solver from generating the same (and hopefully many similar) models again. For good performance it is vital to extract a good reason from the unsatisfiable constraint problems.

2.1 Analysing Unsatisfiable Constraint Problems

In the integer linear programming literature, the problem of detecting reasons for unsatisfiable constraint problems is well known. An optimal reason is called an IIS, or an Infeasible Irreducible System, where feasibility is synonymous with satisfiability.

Definition 2.1 An *Infeasible Irreducible System* (IIS) is a minimal set of infeasible constraints.

This means that a system is an IIS iff

- it is infeasible (unsatisfiable).
- it is not possible to remove any of the constraints from the set without making it feasible (satisfiable).

Any infeasible system contains at least one IIS. Lets look at a few examples. The system

$$\begin{aligned} &(1) && x_1 > 0 \\ &(2) && x_2 - 2x_1 = 1 \\ &(3) && x_2 \leq 1 \end{aligned}$$

is an IIS, since it is infeasible, and it is impossible to remove any of the constraints without making it feasible. The system

$$\begin{aligned} &(1) && x_1 > 0 \\ &(2) && x_2 - 2x_1 = 1 \\ &(3) && x_2 \leq 1 \\ &(4) && x_1 < 0 \end{aligned}$$

on the other hand, is not an IIS. It is possible to remove both 2 and 3 without making the system feasible. Note that if 4 is removed, neither 2 nor 3 can be removed. This means that there are two IISs for this system, $\{1, 2, 3\}$ and $\{1, 4\}$. It is also possible to have an IIS with only one constraint as in this example:

$$2x = 1$$

The system does not have any integer solutions, so it is infeasible, and an IIS.

There are several methods for discovering IISs, both for Linear Programming problems [6,5] and for Integer Linear Programming problems [13]. Most of those designed for Linear Programming will also work for Integer Linear Programming [11]. Here we will use a very simple algorithm. Since in an IIS it is not possible to remove any of the constraints without making the system feasible, we can try to remove the constraints in the problem one by one. If a constraint can be removed without making the system feasible we know that the remaining constraints contain at least one IIS. If not, we know that the constraint we removed last is a member in an IIS and must be replaced. This algorithm is known as *deletion filtering*.

There are frequently several IISs in the constraint problem, and then the order in which the constraints are removed determines which of the IISs will be identified. The order chosen here is the order in which the constraints were created during translation from Lustre. The motivation for this is that we want a reason which is as closely related to the property as possible. The translation is done depth-first in a parse tree with the property at the root, and the ordering chosen seems to be a close approximation of what we would like.

In general, to find an IIS in a constraint problem C the deletion filtering algorithm need to solve one constraint problem for each constraint in C. This can be very expensive, and there are other more efficient algorithms. In particular, the *elastic filtering* algorithm [6,5] is reported to be much more efficient. Despite this, deletion filtering can be shown to work well in this context [11].

3 A Simple Incomplete Constraint Solver

Most of the constraint problems that are generated during search are unsatisfiable. This is not a surprise, since it is enough to find one satisfiable constraint problem to generate a model from the formula. It is therefore important to be able to detect unsatisfiable constraint problems efficiently. Most of these problems turn out to have very simple "conflicts", and a cheap procedure will be able to detect these without incurring the cost of a full blown constraint solver for integer linear programming. This section describes such a cheap procedure that can detect one type of simple unsatisfiable constraint problems, and also find approximations of reasons for these problems.

3.1 Preliminaries

We have a total ordering on all variables $\prec$. A total ordering is one where if $i \neq j$, then either $v_i \prec v_j$ or $v_j \prec v_i$. A constraint is on *normal form* if the greatest common divisor $\gcd(a_1, \ldots, a_n, b)$ of the factors and the right hand side is 1. Any constraint can be rewritten to normal form by dividing every coefficient and the right hand side by $\gcd(a_1, \ldots, a_n, b)$.

An equality is a *definition* if the coefficient of the greatest variable (in the

chosen variable ordering) is 1 or -1. Definitions can be rewritten on the form $x_k = \sum_{i \neq k} a_i x_i + b$.

3.2 The algorithm

The basic idea is that if we have two constraints

$$x < b_1$$

$$x > b_2$$

these two constraints are infeasible iff $b_1 \leq b_2$. An algorithm that uses this fact take one constraint at a time, simplifies it by applying all known definitions and eliminating a number of variables. If the new simplified constraint is a definition, this definition is applied to all known constraints, eliminating one more variable. Then the new constraint is added to the set of known constraints. If any pair of known constraints that are of the form $\pm x \leq b$, contradicts each other, an infeasiblility has been detected. The algorithm can

Algorithm 1 Detect infeasibilities

Require: C is a set of constraints
  $T \leftarrow \emptyset$
  for all $c \in C$ **do**
    Use all definitions in T to simplify c
    if c is a definition **then**
      simplify all constraints in T with the definition c
    end if
    $T \leftarrow T \cup \{c\}$
    if T has a pair $(x \leq b_1, -x \leq b_2)$ where $b_1 < -b_2$ **then**
      return infeasible
    end if
  end for
  return unknown

find explanations of infeasibilities by keeping track of which definitions are used when simplifying constraints. When a pair of conflicting constraints is discovered, the reason is then besides the pair, the definitions that were used, directly or indirectly, to simplify those two constraints to their final form.

When an infeasibility has been detected by the procedure, it is possible to continue adding constraints from the constraint set, perhaps discovering more infeasible subsystems.

For every SAT model, this incomplete procedure can be tried. Only in those cases where it fails to find a conflict will a more expensive procedure have to be used.

4 An Incremental SMT Procedure

The procedure described above has been implemented with an incremental interface, since it will be used for inductive verification. The incremental SAT solver MiniSAT [9] have been extended to a SMT procedure, giving a procedure with an interface described below in C++-like pseudocode.

```
Var addPropositionalVariable();
```

Adds a new propositional variable to the solver.

```
Var addGuardedConstraint(Constraint c);
```

Adds a new constraint to the solver and returns its guard, a propositional variable.

```
void addClause(List< Literal > clause);
```

Adds a clause to the solver as a list of literals.

```
bool solve(List< Literal > assumptions);
```

Solves the problem. The method determines if the set of clauses added so far are satisfiable under the assumption that all literals in assumptions are true. If the formula is satisfiable, a model for it can be retrieved.

Conflict clauses representing reasons for unsatisfiable constraint problems are added to the SAT solver permanently. This is done in part to ensure completeness, and in part for simplicity.

5 Experimental Evaluation

A tool named Rantanplan[4] based on the ideas presented here have been compared with two competitors: NBAC [2], based on abstract interpretation[17], and a modified version of Luke [3] version 0.4beta. Luke is a induction-based program using an eager encoding to a SAT solver. The modification of Luke consists of the SAT solver being exchanged from Satnik to MiniSat[9] for performance reasons.

Rantanplan is based on Luke, and shares much of the code with Luke. The program is available from http://www.dit.unitn.it/~anders/rantanplan/ together with the test suite used for the evaluation.

[2] The version for unbounded integers released on the 3rd of November 2004 is used here.
[3] At the time of writing available at http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html.

Tool	Verified	Alone in verifying
NBAC	96	18
Luke	98	1
Rantanplan	106	1

Table 1
The number of properties verified by the different tools.

5.1 Experimental setup

The tests were performed on a computer with a Intel Xeon 3GHz processor with 1024 kB cache, and 4 GB RAM running Linux. NBAC was run with the following flags: `+eliminput --cudd "-maxmem 128"` The memory limit for CUDD has been increased to 128 MBytes since some examples exceeded the default limit. `+eliminput` is necessary because some properties reason about the input signals in the current time point.

Luke represents integers a vectors of bits using propositional literals. The size of these bit vector can be changed by the user, and for these tests, Luke was run with the default of 16 bit integers. For easy problems, the size of the integers does not affect affects performance enough to change the results. Difficult problems in these experiments are difficult even for very small sizes of the integers, so the choice of bit vector size is not critical.

5.2 Test suite description

The test suite is comprised of a number of Lustre programs together with properties on the programs. A test is one program/property pair. Since many programs have several properties, the same program is used in several tests. For programs which has more than one property, there is also a test where the property is the conjunction of all properties on the program. The test suite consists entirely of academic examples of Lustre programs. Several are for instance models of cache coherence protocols [8].

The tests are chosen such that they can be verified with any tool, given enough time. Those tests which can not be verified with one or more tools have been eliminated. An overview of the results for all test is given in table 1. It should be noted that of the 127 tests in the test suite, 7 were not verified by any tool.

Of these 127 tests, only 72 can be used for comparisons of execution times. There are three reasons why tests have been excluded:

- NBAC can not handle invalid properties. A small comparison between Luke and Rantanplan for invalid properties can be found in section 5.3.1.
- For both Luke and Rantanplan, some properties of some Lustre programs can not be verified in isolation, but only together with all other properties

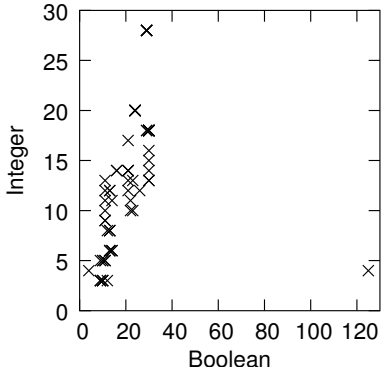

Fig. 1. The number of boolean and integer variables for the examples.

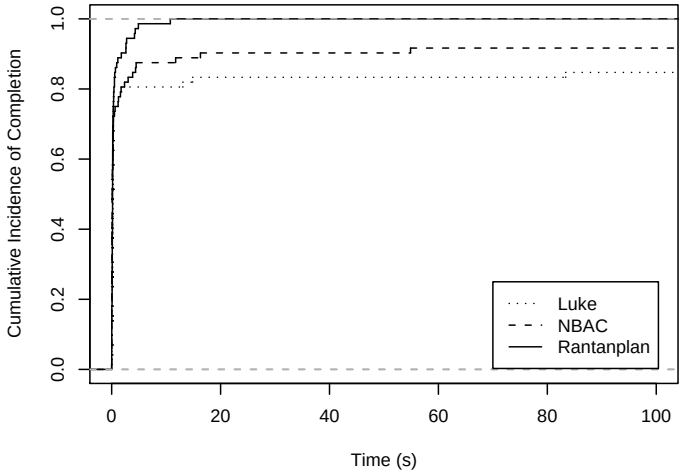

Fig. 2. Results for all Lustre examples.

of that program. NBAC on the other hand can only verify one property at a time, making it difficult to compare performance.

- Some properties are valid for unbounded integers and invalid for bounded integers as modeled in Luke.

The remaining 72 tests are of varying complexity. All tests are fairly small, as can be seen in figure 1, which shows the number of integer and boolean variables for the tests. The tests were run with a timeout of 100 seconds.

The results are presented here in the form of cumulative incidences of completion. This is the probability that a randomly chosen test taken from the test suite terminates as a function of time. Another way of interpreting the data is that it is the ratio of terminated tests as a function of time. As can be seen in figure 2, there is a group of tests where both NBAC and Luke struggles.

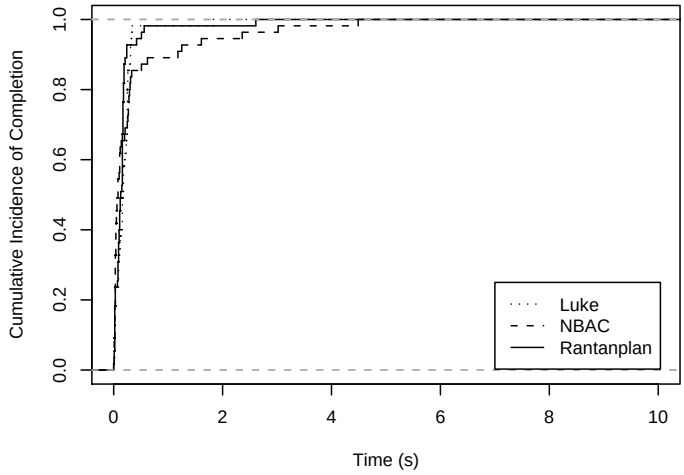

Fig. 3. Results for easy Lustre examples.

In this group we find tests with more complex arithmetic. These tests include arithmetic expressions with the sum of two or more integer variables, whereas the rest only contains simple arithmetic expression where one of the operands is a constant.

Removing the examples which takes more than 10 seconds in either NBAC or Luke leaves the data in figure 3.

5.3 Explanation of the differences

The examples where NBAC and Luke have trouble all have one thing in common: They use the sum of two or more variables in expressions. All other examples only add or subtract to variables by a constant. NBAC also have trouble with conjunctions of several properties, where induction based methods improves for stronger properties. The examples which can not be verified with all (three) tools are not part of the test suite, and this may bias the tests. The original test suite consisted of 127 tests.

5.3.1 Invalid properties

NBAC does not have built-in support for discovering that a property is invalid. A tool NBAC2LUCKY will soon be publicly available to interface to the symbolic simulator Lucky [16] to aid in finding counter-examples.

Induction is complete for invalid properties, so both Luke and Rantanplan can find the shortest possible counter-example for an invalid property. Luke outperforms Rantanplan on invalid properties with longer counter-examples, as can be seen in figure 4. The test in the figure has counter-examples ranging from 2 to 7 steps.

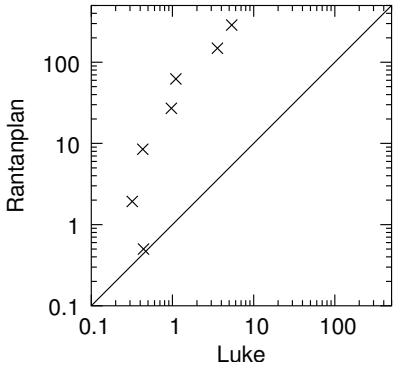

Fig. 4. Comparison of execution times between Luke and Rantanplan on invalid properties with increasingly longer counter-examples

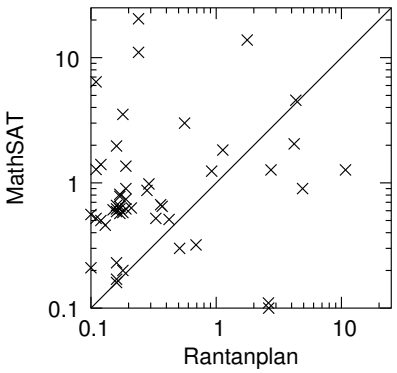

Fig. 5. Comparison of execution times (in seconds) between Rantanplan and MathSAT 3.2.1

5.4 Comparison with MathSAT

As there are several highly advanced SMT solvers available, it would be interesting to make a comparison with at least one of those. Therefore, a feature has been added to Rantanplan allowing the tool to output SMT formulas in the MathSAT format. The formulas created are the base and step cases needed to verify the property in question. These formulas have been run with MathSAT version 3.2.1, and an overview of the results can be seen in figure 5. Each point in the scatter plot is a pair of execution times for Rantanplan and MathSAT respectively. The execution times of MathSAT is the total execution time for all base and step cases which are solved by Rantanplan in order to verify a property. A possible explanation of why such a simple procedure as described here sometimes has a better performance would be that MathSAT is not an incremental decision procedure, and therefore would have to "relearn" conflicts for each new case.

6 Conclusions and Future Work

We have seen that an incremental SMT procedure can be constructed with very simple means, and still perform well for inductive verification of Lustre programs in comparison with other methods. There is a class of problems with larger linear integer Lustre expressions where our approach clearly outperforms other methods.

In comparison with other SMT decision procedures the procedure described here is not very advanced. That the performance is still comparable to NBAC and Luke can be attributed to three key areas where special care have been taken. The first is the translation from Lustre to the logic. Being able to only take the relevant constraints into consideration for each SAT model gives a huge performance boost in this tool. Next is the heuristic for discovering reasons for unsatisfiability in constraint problem. There is typically several reasons to choose from, and their ability to prune the search space varies a great deal. Lastly the incomplete procedure for detection of unsatisfiable constraint problems makes it much more efficient.

There are several improvements which can be expected to improved performance further. These are among others early pruning, which is checking satisfiability of partial propositional truth assignments. For cases such as linear arithmetic where this may be expensive, there is *weakened early pruning* using an incomplete procedure to catch most usatisfiable constraint problems. Another idea often used is preprocessing of the constraints, trying to discover unsatisfiable combinations of constraints before the search starts. These and other more recent innovations are sucessfully used in other SMT procedures, and can be expected to be beneficial here as well.

References

[1] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[2] P. Bjesse and K. Claessen. SAT-based Verification without State Space Traversal. In *Proceedings of FMCAD 2000*, 2000.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS'05*, 2005.

[4] M. D. Bévère. *Sur la piste des Dalton*. Dargaud Lucky Productions, 1960.

[5] J. W. Chinneck. Finding a Useful Subset of Constraints for Analysis in an Infeasible Linear Program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[6] J. W. Chinneck and E. W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[7] D. Deharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *Proc. SEFM'03*. IEEE Computer Society Press, 2003.

[8] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proccedings of CAV 2000*, 2000.

[9] N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[10] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. In *Proceedings of the First International Workshop on Bounded Model Checking*, 2003.

[11] A. Franzén. Combining SAT Solving and Integer Programming for Inductive Verification of Lustre Programs. Master's thesis, Chalmers University of Technology, 2004.

[12] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *16th International Conference on Computer Aided Verification (CAV)*, Boston (USA), July 2004.

[13] O. Guieu and J. W. Chinneck. Analyzing Infeasible Mixed-Integer and Integer Linear Programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[15] N. Halbwachs and P. Raymond. A Tutorial of Lustre. 2002.

[16] E. Jahier and P. Raymond. The Lucky language Reference Manual. Technical Report TR-2004-6, Verimag.

[17] B. Jeannet. Dynamic Partitioning in Linear Relation Analysis. Application to the Verification of Synchronous Programs. *Formal Methods in System Design*, 23(1):5–37, 2003.

[18] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Lecture Notes in Computer Science*, volume 1954. Springer Verlag, 2000.

[19] H. M. Sheini and K. A. Sakallah. A SAT-based Decision Procedure for Mixed Logical/Integer Linear Problems. In *Lecture Notes in Computer Science*, volume 3524, pages 320–335, 2005.

Adaptive Application of SAT Solving Techniques

Ohad Shacham [1,2] Karen Yorav [1,2]

IBM Haifa Research Laboratory
Israel

Abstract

New heuristics and strategies have enabled major advancements in SAT solving in recent years. However, experimentation has shown that there is no winning solution that works in all cases. A degradation of orders of magnitude can be observed if the wrong heuristic is chosen. The problem is that it is impossible to know, in advance, which heuristics are best for a given problem. Consequently, many ideas - those that turn out to be useful for a small subset of the cases, but significantly increase run times on most others - are discarded.

We propose the notion of *Adaptive Solving* as a possible solution to this problem. In our framework, the SAT solver monitors the effectiveness of the search on-the-fly using a *Performance Metric*. The metric gives a score according to its assessment of the search progress. Based on this score, one or more heuristics are turned on or off. The goal is to use a specific heuristic or strategy when it is advantageous, and turn it off when it is not, before it does too much damage. We suggest several possible metrics, and compare their effectiveness. Our adaptive solver achieves significant speedups on a large set of examples. We also show that applying different heuristics on different parts of the search space can improve run times even beyond what can be achieved by the best heuristic on its own.

Key words: SAT solving

1 Introduction

Recent years have seen great amounts of research on SAT solving [6,12,15]. The problem is interesting theoretically, as well as important for practical reasons. The high capabilities of advanced solvers has encouraged their use in various fields such as verification, artificial intelligence, CAD, and more. This, in turn, created great incentive to invest in research of SAT solving techniques.

[1] Email: {ohads,yorav}@il.ibm.com
[2] This research is supported in part by EU contract FP6-IST-507219 (PROSYD)

Our perspective on SAT solving comes from its use in *Bounded Model Checking* [3] (BMC), where the verification problem of a hardware design is translated into a Boolean formula such that a satisfying assignment, if one exists, represents a counterexample. Most tools that implement this framework are based on DPLL-style SAT solvers. Although the ideas presented in this work are general, and may easily be applied to other types of SAT solvers, our results are tuned for BMC instances. We believe the method is most efficient when applied to instances that have internal structure.

Modern SAT solvers rely heavily on various heuristics and strategies such as decision heuristics, restart strategies, learning strategies, clause deletion strategies, etc [2,6,11,12,15,16]. However, many ideas that seem appealing in theory turn out not to perform well in practice, decreasing the run time on a few examples while increasing it on most others. As a result, these ideas are discarded. Even successful heuristics are not useful in all cases, but it is impossible to know beforehand which heuristics are most suitable for a given example.

In this paper we propose the concept of *Adaptive Solving*. Adaptive Solving optimizes the way different strategies are used, by applying them when they are useful and turning them off when they are not. The adaptive solver tracks the performance of the search and evaluates it using a *Performance Metric*. Whenever the search seems not to be progressing well enough, it changes run-time parameters by enabling or disabling heuristics. In this way the adaptive solver is capable of making use of heuristics that do not work well in all cases.

We propose several metrics to be used in adaptive solving. These metrics are easy to compute and incur a negligible overhead. They track different aspects of the search and give a score accordingly. We compare their effectiveness and present some insights to their use.

Our BMC tool is part of RuleBasePE [9], a parallel model checker developed at the IBM Haifa Research Laboratory. This tool uses our in-house solver called ***Mage***. We have implemented an adaptive version of Mage and used it on a large number of examples. Results show that adaptive solving reduces the overall run time by more than a third, and achieves speedups of up to 12x on single examples.

Naturally, there are examples for which run time is increased as a result of enabling or disabling a heuristic. The overall reduction is achieved by significantly reducing run times on many examples, while increasing the run time of others to a lesser degree. Results show that in general speedups are better for larger examples, making the method highly scalable. Furthermore, we show that even when a heuristic performs badly for a certain example, applying it on parts of the search may give better results than not using it at all. This means that adaptive application of heuristics gives performance improvements over the best heuristic on its own.

Our implementation is an initial experiment in Adaptive Solving. It controls only one heuristic, and uses only one metric at a time, and yet it achieved

impressive speedups.

Related Work
Previous attempts have been made at assessing the progress of the solver's search for a satisfiable assignment. Aloul, Sierawski, and Sakallah view the conjunction of conflict clauses as representing the space that is yet to be covered, with each added conflict clause reducing this space until it is empty. Their Satometer [1] tool keeps a BDD representation of this conjunction. Of course, the exact space cannot be calculated, so the tool uses an approximation. The drawback of this approach is its huge overhead - both in space and in computation, which prevents it from being used as a performance metric.

Another related work is presented by Herbstritt and Becker in [7], where decision heuristics are chosen dynamically, according to a set of probability functions. The probabilities are changed according to several criteria. Our work is more general because we address any run-time parameter of the solver, not just the decision heuristic.

Nudelman et al [14] use machine learning to identify features of CNFs. Using a large training set they learn the correlation between the hardness of the problem and the result of different kinds of analyses of the CNF. Their SATzilla tool [13] profiles the run times of several different solvers using the same training set and features. When a new problem is to be solved, the tool computes the different features and then chooses the solver predicted to have the least run time. However, the features they use to choose the solver are not applicable for us, because they are more relevant for random instances than for structured ones, and because they need to be computed beforehand.

In Lagoudakis and Littman's work [10], decision heuristics are chosen according to a value function, which is calculated on the current state of the search. The value function is created beforehand, using a training set. The training set must be a significantly large set of examples that are similar in some sense to the CNFs we want to solve. Using a training set is problematic both because it incurs a high overhead, and because it is difficult to generate an adequate training set, especially in the setting of SAT-based verification.

The remainder of the paper is structured as follows. Section 2 provides an overview of DPLL-style SAT solver algorithms. Section 3 presents a variety of performance metrics. Section 4 defines Adaptive Solving. Finally, Section 5 shows the experimental results, and Section 6 presents our conclusions.

2 Background on SAT Solving

Given a Boolean formula F over a set of variables V, the task of the SAT solver is to find an assignment to all variables in V such that F is satisfied. The formula is given in *Conjunctive Normal Form* (CNF). A CNF formula is a conjunction of *clauses*, where each clause is a disjunction of *literals*. A *literal* is an instance of a variable, x, or its negation, $\overline{x}$.

A literal may have one of three values: *true*, *false*, or *undef* (undefined). A clause in which one literal is undefined and all the rest are *false* is called a *unit clause*. Such a clause forces an assignment of *true* to the undefined literal, as this is the only way to satisfy the formula.

2.1 SAT Solving Algorithms

Our implementation is geared towards DPLL-style solvers [5] with conflict clause learning and non-chronological backtracking [2,11], although the adaptive solving concept can be applied to other solving schemes. We give a brief description of DPLL with learning. For a more thorough discussion see [17].

```
while(1) {
  if (decide_next_branch())
    while (deduce() == CONFLICT) {
      blevel = analyze_conflicts();
      if (blevel == 0) return UNSAT;
      else bactrack(blevel);
    }
  else
    return SAT;
}
```

Fig. 1. Basic DPLL algorithm with learning

Figure 1 gives a bird's eye view of a DPLL algorithm with learning. The algorithm iteratively chooses a value for a variable (`decide_next_branch()`). If all variables become assigned the algorithm halts, and outputs a satisfying assignment. Otherwise, the implications of this assignment are carried through by the `deduce()` function. If `deduce()` reveals a conflict, the reason for the conflict is analyzed and a conflict clause is added to the database. The conflict clause summarizes the combination of values that lead to the conflict and prevents this combination from being assigned a second time. The function `analyze_conflicts()` returns a decision level to which the algorithm backtracks. If this is level 0, it means that there exists a conflict even without a single decision, which means that the formula is unsatisfiable. Otherwise, the algorithm backtracks and continues the search.

2.2 Decisions

The `decide_next_branch()` function chooses a variable that is undefined and assigns to it either *true* or *false*. This is called a decision, and the variable is called the decision variable. The algorithm that controls the way this choice is made is the *decision heuristic*. Each decision is associated with a number, called the *decision level*. All the implications that result from one decision are associated with the same decision level. When constant values are revealed, they are associated with decision level zero.

2.3 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) is the process of propagating the effect of an assignment. This is the task of the function `deduce()` in Figure 1. Each assignment may cause several clauses to become unit. Each unit clause implies an assignment, which may in turn result in more unit clauses. During BCP, this process is iterated until no further assignments can be implied.

Since modern solvers spend roughly 80% of their time carrying out BCP, it is crucial that this process be implemented efficiently. The technique used in Mage, as in zChaff and others, is to mark two literals in every clause as *watched literals*. The rational is that a clause of length n (with $n \geq 2$) can become a unit clause only after $n - 1$ of its literals have been assigned *false*. Only unit clauses can cause implications, so as long as the two watched literals of a clause are undefined (or *true*) this clause is not unit, and there is no need to examine it during BCP. Whenever a literal l is assigned *false*, all clauses in which l is watched are examined to see whether they have become unit.

2.4 Clause Learning and Non-Chronological Backtracking

A conflict happens when BCP propagates a certain assignment and discovers a clause with all its literals set to *false*. During conflict analysis, the chain of implications that resulted in the conflict is analyzed, and the reason for the conflict is summarized in a *conflict clause*. This clause describes a combination of assignments that should not be repeated as they are conflicting. The conflict clause is added to the clause database, thus pruning the search space that remains to be traversed.

3 Performance Metrics

The performance of a SAT solver is measured by the time it takes to solve a given instance. In order to evaluate whether a certain heuristic or strategy is beneficial, the overall run time on many different formulas is compared. The reality is that even for the best heuristics, there are examples on which they do not work well. The role of the performance metric is to assess the compatibility of the solver settings for a specific example during the run.

We propose the following requirements on performance metrics:

(i) The metric can be evaluated during the solver's run.

(ii) It can be calculated efficiently, i.e. with low overhead, and with minimal additional space consumption.

(iii) The metric gives a score that (roughly) corresponds to the effectiveness of the search.

The nature of the SAT problem is such that it is unrealistic to expect to find a metric that will give a perfect correlation to the end result. However, based on our understanding of the way the solver operates, we suggest several

candidates, which are listed below. Each metric is evaluated on a *sample* of the run, consisting of a constant number of decisions.

3.1 Decision Level

When a decision causes a conflict, the solver backtracks to a previous decision level and cancels all the assignments made in between. In this case the decision level of the next decision will be some smaller number. Otherwise, if there was no conflict, the decision level increments by one.

The **DL** metric looks at the average decision level in a single sample (in our case - 2048 decisions). The solver reaches high decision levels when it makes a large number of decisions without conflicts, or when the conflicts do not set it back by much. As a result many variables keep their value for long periods of time. This could mean that the solver spends significant amounts of time searching in a small part of the state space. On the other hand, a low average may indicate a high conflict rate. Since each conflict clause restricts the space that remains to be searched, a high conflict rate is a good sign. This leads us to expect that an efficient run is one in which the average decision level is relatively low.

It should be noted that the average decision level is greatly influenced by the internal structure of the solver and the chosen decision heuristic. When experimenting with this metric, we found that the average decision level in zChaff was, in general, twice as high as the average level for Mage when running on the same formula.

3.2 Conflict Clause Size

As mentioned earlier, every conflict clause that is added is a constraint that reduces the state space that remains to be searched. In general, smaller conflict clauses are capable of posing a greater restriction. So, although it is possible for a specific small clause to be less useful than a very large one, in general we expect that very small conflict clauses advance the search more rapidly towards its goal.

Our second metric, **CCS**, is the average length of all the conflict clauses that were learned in a sample - the smaller the better. Note that the score does not explicitly reflect the actual number of conflict clauses, which could be a metric on its own.

3.3 Binary Conflict Clauses

Some Solving strategies emphasize preference towards short conflict clauses [4] and binary clauses in particular [15]. This makes sense in light of the fact that these clauses have the highest potential of generating implications (a single assignment makes the clause unit). We therefore expect that adding many binary clauses to the database greatly advances the search.

The **BIN** metric measures the percentage of binary conflict clauses out of the total number of conflict clauses learned in a given sample.

We have also considered looking at ternary clauses as a part of this metric. However, extensive experimentation revealed that the percentage of ternary conflict clauses is linearly correlated to the percentage of binary clauses. In all of our examples, these two numbers are almost equal, and they increase and decrease in the same manner from one sample to the next. We concluded that there is no added benefit in tracking ternary conflict clauses.

3.4 BCP Ratio

When a watched literal l in a clause c becomes $false$, the BCP process must go over the literals in c and look for a new watched literal. In the worst case scenario, all the literals in c are examined. The **BCP** metric measures the ratio between the number of literals examined (all together) and the number of clauses visited, i.e., it calculates the average number of literals examined per clause. This ratio is indicative of the speed at which implications are carried out. Since the BCP operation is the major part of the computation, it is important to keep this number low.

3.5 Unary Clauses Learning

The `analyze_conflicts()` function is capable of producing unary conflict clauses. This amounts to learning the value some variable must have in order to satisfy the formula. The variable is then assigned a permanent value. When this happens, the algorithm backtracks to decision level zero and applies BCP to discover all implications of this assignment. Any assignment resulting from this BCP process is also permanent.

The **UNARY** metric tracks the rate at which permanent values are assigned. It gives the number of such values assigned in the last sample. An examination of the behavior of this metric reveals that learning happens in bursts. Typically, there are extended periods of time with little or no learning, and then suddenly tens or even hundreds of variables are assigned a value.

4 Adaptive Solving

As mentioned earlier, there is no one heuristic that works well in all cases, whether it is a decision heuristic or any other heuristic that is used during the search. Our solution is an *Adaptive Solver* that is capable of adapting its run-time parameters to the specific CNF it is solving. During the search, the solver looks for signs that the run-time parameters with which it is running are not optimal, and changes them on-the-fly.

An Adaptive Solver works according to the following scheme:

- The run is divided into *samples*, where each sample consists of the computation performed during a certain number of decisions.
- At the end of each sample, a *performance metric* is used to evaluate the effectiveness of the search in this sample.
- A *switching condition* decides whether the solver is progressing.
- If the switching condition evaluates to *true*, one or more *parameters* of the run are changed. We call this *a switch*.

The specifics of an adaptive solving algorithm include the size of a sample, the choice of a performance metric, the choice of parameters to change, and the condition for switching. The possibilities for all these are endless. In the end, the right choice of these elements can make the difference between success and failure. Furthermore, the choices for each element depend on the specifics of the SAT solver implementation. There are no clear cut rules for building an adaptive solver. The rest of this section describes, and motivates, some of the choices made for Mage, and the insights gained from experimentation.

4.1 The Parameter

The choice of parameters to switch is important. The idea is to choose a parameter that has a high impact on the run time. At the same time, there is no point in adaptively switching a parameter that is always useful. Luckily, there is no shortage of such options. There are numerous heuristics and strategies that are not used in practice because they are beneficial only for some examples, but detrimental for most.

In our adaptive implementation of Mage, we chose the `-sign` option as the parameter to control. As this is an initial experimentation in Adaptive Solving, only one aspect of the SAT solving algorithm was controlled adaptively. This option controls the way a value is chosen for a decision variable in the `decide_next_branch()` function. Normally, after choosing a variable to decide upon, the function chooses whether to assign it *true* or *false* by examining the scores of the corresponding literals. The default is to assign *true* to the literal with the highest score. Activating the `-sign` option will make `decide_next_branch()` assign *true* to the literal with the lower score.

Experimentation shows that in general, it is better to choose the literal with the higher score. In most examples this will result in shorter run times. However, for some examples, choosing the lower scored literal can result in a speedup of up to 4`x` (see results in Section 5).

4.2 Switching Conditions

We implemented a mechanism to compute all of the metrics mentioned in Section 3. The sample size we chose is 2048 decisions. When using a smaller sample we found that the metric score did not stabilize, and another switch

would occur too soon [3].

For each of the DL, CCS, BIN, and BCP metrics, we chose an initial bound, so that a switch is made when the metric score exceeds the bound. The initial bound was chosen by running a large number of examples without adaptive solving, and inspecting the scores each metric produces. After the runs completed, we saw which were the best in terms of run times. We looked at the average score of the "bad" runs, and the average score of the "good" ones, and chose the initial bound for each metric to be some number in between.

In the case of the UNARY metric a switch is made if the number of added permanent values is low over a period of several samples. Because of the effect mentioned earlier, in which permanent values are added in bursts, the condition for this metric was set to: "perform a switch if in 14 out of the last 16 samples the number of permanent values added was low" (less than 3). Again, this scheme was developed by examining the runs of good and bad examples. We noticed, for example, that even for the worst cases there may be one or two added values in each sample, which is why we do not require this number to be zero.

Initial experimentation revealed that for some of the hard problems the metric scores were consistently high, and the adaptive solver was switching parameters throughout the run. This caused a significant increase in run times. It seems that in order for a heuristic to be effective, it needs to run for a certain period of time without interruption. Consequently, we placed several mechanisms to prevent the adaptive solver from switching too often:

- When a switch is made the metric bound is incremented (or decremented), so that in order for another switch to occur the metric score will have to be slightly worse than it was in the last switch.
- After each switch, further switching is prevented during the next 20 samples.
- A limit is placed on the number of switches allowed during a single run.

5 Experimental Results

We conducted extensive experimentation on our adaptive version of Mage. Our benchmark suite includes 50 examples from the IBM Benchmarks Suite [8]. This is a collection of CNF files that originate from the verification of various industrial designs using SAT-based BMC. The benchmark is very diverse, with both long and short examples, satisfiable and unsatisfiable, various depths, etc.

Our Adaptive Solver enables switching only after $20,000$ decisions have been executed, so that no switches are made for easy problems. The benchmarking suite includes only examples that require more than $20,000$ decisions. We also made sure that the examples we consider do not abort because of time-out. This is done so that the time we choose to time out on will not influence

[3] The sample size is an exponent of 2 because this makes the implementation more efficient.

the speedup results. Although it may seem impressive to report that there are several examples on which the native version timed-out while the adaptive version succeeded, in reality this only depends on the time-out constant. Instead of reducing this constant to generate such "impressive" examples, we chose to enlarge it to the point where all of the examples we want to run do not time-out. This required a relatively high timeout of $10,000$ seconds. The only two runs that time out on $10,000$, are examples that run with the `-sign` option and no Adaptive Solving. This version does not influence the analysis of the results for the adaptive algorithm, it is merely used to demonstrate that the `-sign` option is overall detrimental.

The analysis of the results is done by comparing the overall speedup, i.e., the speedup on the sum of the run times of all examples. This number is indicative of the effect Adaptive Solving has on a large number of examples of various sizes, such as in the case of a full verification project. This analysis places more weight on the larger examples. This is suitable for our experimentation, because our goal is to reduce the run times of large examples without hurting the small ones too much, thus reducing the overall time needed for a verification project to be completed. Calculating the average speedup, for example, would place more emphasis on reducing a two-second run to one second, than adding an hour to an example that runs two hours. This may be suitable for theoretical analysis, but it is not suitable for an industrial tool. Because this is a prototype implementation of a new idea, however, we also give the minimum and maximum speedups. We consider these an indication to the potential of Adaptive Solving.

Our experimentation was conducted on an Intel Pentium 4, with a single 2GHz CPU, 1GB RAM, running Linux. Tables 2, 3 give the run time results for all 50 examples, in seconds. It compares the run times between seven versions. **Native** is Mage running with its default parameters and no adaptive algorithms. In particular, the `-sign` option is not used, so in all decisions the sign with the highest score is chosen. **Sign** is Mage running with the `-sign` option all of the time. Versions **DL**, **CCS**, **BIN**, **BCP**, and **UNARY** correspond to adaptive versions each using the metric implied by its name (see Section 3). The Native and Sign versions do not calculate any of the metrics. Initial experimentation showed that the overhead incurred by the computation of the metric scores is negligible (only a few seconds even for the largest examples). Because the difference is so small we omit the results for a version that computes all metrics but does not apply adaptive solving.

Table 1 gives a summary of the results. In this table the "Time" rows display the sum of run times on all the examples in the benchmark (in seconds). The "Speedup" for each version is the runtime of Native divided by the runtime of that version. The "Min" and "Max" rows show the minimum and maximum speedups achieved by each version on a single example. The results for satisfiable and unsatisfiable examples are given separately, and the "ALL" section summarizes all the examples.

Table 1 shows that BIN and UNARY are the best metrics, giving speedups of 1.6 and 1.5 respectively. The BCP and CCS versions give modest speedups, and DL has a negligible speedup. For the CCS, BIN, and UNARY versions, the speedup is better on SAT instances than on UNSAT. On SAT instances alone, UNARY gives a $2x$ speedup, while on the UNSAT instances, there is hardly any gain. On the other hand, the BCP version works better on UNSAT instances. The `-sign` option is, indeed, not recommended as a default option, since it significantly increases the overall run time.

Examining the minimum and maximum speedups reveals how the overall speedup is achieved – by significantly reducing run times on some examples and only slightly increasing run times on others. For example, the worst damage the BIN version causes is a speedup of 0.75 (which equals to an increase of about a third), while its best performance is almost $12x$ faster. Note that all of the examples in the benchmark suite are non-trivial. The best speedup was achieved on an example that runs 3821 seconds on the Native version, and 325 seconds with the BIN version.

A phenomenon we encountered, and can be seen in tables 2, 3 is that in many cases the adaptive version performs better than either Native or Sign. From this we learn that different sub-spaces of the search space require different settings. This encourages us that Adaptive Solving has great potential.

	Version	`Native`	`Sign`	`DL`	`CCS`	`BIN`	`BCP`	`UNARY`
UNSAT	Time	8662	14579	8609	7726	6702	7057	7933
	Speedup	-	0.594	1.006	1.121	1.292	1.227	1.091
	Min	-	0.097	0.771	0.874	0.831	0.830	0.913
	Max	-	4.354	1.641	3.878	4.042	2.951	4.017
SAT	Time	14955	25256	13067	9228	8269	12637	7313
	Speedup	-	0.592	1.144	1.620	1.808	1.157	2.044
	Min	-	0.067	0.168	0.509	0.751	0.411	0.523
	Max	-	4.437	3.900	5.287	11.749	1.326	5.900
ALL	Time	23618	39835	21676	16954	14971	19695	15247
	Speedup	-	0.593	1.089	1.393	1.578	1.182	1.549
	Min	-	0.067	0.168	0.509	0.751	0.411	0.523
	Max	-	4.437	3.900	5.287	**11.749**	2.951	5.900

Table 1
Summary of run time results for all adaptive versions

6 Conclusions

We view the Adaptive Solving algorithm presented here as a starting point rather than a finished product. As mentioned before, there are many design decisions in the implementation of an Adaptive Solver that can make a difference in its performance. Our choices are by no means guaranteed to be the

best possible.

Nevertheless, the prototype algorithm was able to achieve up to 12x speedup in run times on satisfiable examples, up to 4x speedup on unsatisfiable examples, and an overall 1.6 speedup on the whole benchmark suite. The speedup on the sum of all run times is particularly significant in the setting of SAT-based verification, since it represents the impact Adaptive Solving can have on a whole verification project. Our results show that the overall time needed for the project can be reduced, by having some of the runs finish significantly faster and others slightly slower. The result is a verification effort that is completed in less time, and reveals bugs much faster.

We showed that the `-sign` option has an overall detrimental effect on run times. Although there are some examples for which this option is useful, when using it on all of our examples the overall run time is much larger. It is impossible to predict beforehand which examples will benefit from this option. For this reason it has not been used in practice until now. The Adaptive Solver is thus capable of making the best out of a heuristic that overall did not prove beneficial. Our implementation controlled only one such heuristic, but there are many others that could be used. We plan to investigate how to combine this strength on multiple options.

An interesting phenomenon is that on some examples, although the `-sign` option performs badly, using it on parts of the search space gave better results than not using it at all. This shows that different sub-spaces require different approaches, and clearly demonstrates that the potential of Adaptive Solving is greater than that of the parameters it controls.

As for the performance metrics – we continue to search for better metrics. We have discovered that some metrics perform better on satisfiable instances, while others are better for unsatisfiable instances. This implies that a combination of metrics may be more beneficial.

The details of the adaptive algorithm need to be tuned according to the specific implementation of the solver. The organization of the database and the decision heuristics used by a solver influence the right choices for the adaptive algorithm. This means that implementing the exact same algorithm on different solvers, may yield different results.

There are many directions that require further research. Finding the best metric and parameters to use is crucial. The algorithm used to determine when to switch options has also not been perfected. Beyond this, we would also like to experiment with incorporating learning algorithms into the process.

References

[1] F. A. Aloul, B. D. Sierawski, and K. A. Sakallah. Satometer: how much have we searched? In *DAC*, pages 737–742. ACM Press, 2002.

[2] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Conf. on AI*, pages 203–208, 1997.

	Native	Sign	DL	CCS	BIN	BCP	UNARY
IBM_02_1_1_cycle_45	26.67	29.11	23.67	24.79	26.84	26.76	26.62
IBM_02_1_1_cycle_50	26.87	23.51	24.8	27.11	27.11	27.17	26.98
IBM_04_cycle_45	39.95	30.15	38.73	31.08	40.01	30.98	39.82
IBM_05_cycle_45	19.28	32.71	28.25	19.21	19.13	23.06	19.1
IBM_05_cycle_50	23.58	42.47	30.48	23.56	23.53	30.88	23.53
IBM_06_cycle_45	47.62	11.32	47.24	41.52	47.5	47.42	47.43
IBM_07_cycle_10	62.35	341.05	40.29	44.42	43.65	46.62	42.59
IBM_07_cycle_15	24.28	18.5	19.63	21.81	16.4	16.69	19.49
IBM_07_cycle_20	24.57	30.14	20.17	20.18	19.19	19.28	20.64
IBM_07_cycle_25	24.74	48.81	22.02	26.55	21.42	23.67	18.41
IBM_07_cycle_30	25.14	50.95	17.43	22.63	18.91	18.32	15.42
IBM_07_cycle_35	25.53	22.8	28.7	22.02	14.01	15.57	17.45
IBM_07_cycle_40	25.9	27.19	20.88	19.99	19.68	21.96	17.8
IBM_07_cycle_45	26.02	192.98	15.86	22.02	28.04	29.16	26.58
IBM_07_cycle_50	26.33	19.4	25.45	22.9	16.45	16.41	24
IBM_11_1_cycle_45	1140.99	1232.29	679.36	1175.36	952.89	1054.36	894.69
IBM_14_2_cycle_25	25.62	22.42	25.88	19.43	25.87	20.7	25.68
IBM_14_2_cycle_30	51.18	58.86	66.39	46.96	43.52	37.62	50.96
IBM_14_2_cycle_50	669.84	6881.23	668.32	535.39	618.71	593.9	684.73
IBM_17_1_2_cycle_40	16.23	8.89	19.14	16.25	16.19	16.17	16.3
IBM_17_1_2_cycle_45	19.38	7.79	24.37	19.39	19.92	19.37	19.32
IBM_17_1_2_cycle_50	15.98	11.64	15.77	16.15	19.23	16.03	15.29
IBM_19_cycle_35	37.96	32.8	30.09	51.42	43.29	37.8	37.71
IBM_19_cycle_40	69.01	120.82	91.02	60.07	84.79	70.69	70.41
IBM_19_cycle_45	112.26	120.35	124.43	220.46	144.64	112.29	155.17
IBM_19_cycle_50	283.95	194.38	401.87	215.21	310.72	341.76	542.8

Table 2
Run times of the adaptive solver versions on each one of the CNF instances.

[3] A. Biere, A. Cimatti, E. Clark, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.

[4] E. Carvalho and J. Marques-Silva. Using rewarding mechanisms for improving branching heuristics. In *SAT*, pages 275–280, May 2004.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[6] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT solver. In *DATE*, pages 142–149, 2002.

[7] M. Herbstritt and B. Becker. Conflict-based selection of branching rules. In *SAT*, pages 441–451, May 2003.

[8] IBM HRL. IBM formal verification benchmark library, 2004. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html.

[9] IBM. Rulebase Parallel Edition, 2004. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html.

	Native	Sign	DL	CCS	BIN	BCP	UNARY
IBM_20_cycle_25	73.59	75.19	73.32	67.84	73.88	88.68	73.86
IBM_20_cycle_30	378.49	313.49	378.6	304.09	378.23	366.73	377.86
IBM_20_cycle_40	2915.99	2193.85	2913.52	3338.03	2098.47	2612.28	3193.65
IBM_20_cycle_45	2262.22	1384.75	1584.61	4376.08	1495.49	1793.88	2104.08
IBM_20_cycle_50	5688.42	1281.89	1458.45	1075.95	3400.8	4287.91	964.08
IBM_21_cycle_35	30.56	24.51	28.6	35.64	30.45	30.47	30.38
IBM_21_cycle_40	97.71	73.52	90.42	92.78	97.43	97.39	97.09
IBM_21_cycle_45	321.15	221.58	230.48	198.86	235.65	322.9	320.48
IBM_21_cycle_50	207.1	335.21	216.49	211.32	275.71	207.17	270.92
IBM_22_cycle_35	38.34	41.92	47.13	40.96	38.24	37.13	38.18
IBM_22_cycle_40	115.97	155.49	124.63	113.16	116.41	121.43	116.21
IBM_22_cycle_45	368.1	612.17	400.4	378.86	366.91	443.05	371.93
IBM_27_cycle_45	12.83	21.72	19.23	18.31	12.73	19.67	12.68
IBM_28_cycle_30	21.22	18.12	21.11	31.1	21.21	21.31	21.19
IBM_28_cycle_40	21.72	21.41	27.08	39.31	21.74	52.79	21.73
IBM_28_cycle_45	22.25	55.86	28.96	25.27	22.29	22.26	22.14
IBM_29_cycle_15	305.51	149.21	283.99	176.02	173.99	183.16	194.88
IBM_29_cycle_20	1828.63	1792.38	1817.36	1745.12	1948.38	1542.37	1764.72
IBM_29_cycle_30	3821.98	10000	3875.55	1013.73	325.29	3505.24	859.39
IBM_29_cycle_50	673.73	10000	4015.01	647.92	663.63	815.04	758.6
IBM_new_2_cycle_20	214.05	152.21	213.85	127.8	69.35	92.03	207.56
IBM_new_2_cycle_25	916.42	1020.32	906.14	236.31	226.7	310.5	228.15
IBM_new_5_cycle_20	137.81	31.65	118.62	122.9	75.56	124.26	80.63
IBM_new_6_cycle_20	252.53	245.97	252.3	146.51	140.69	170.39	217.72

Table 3
Run times of the adaptive solver versions on each one of the CNF instances.

[10] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *SAT*, volume 9, June 2001.

[11] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.

[13] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. Satzilla: An algorithm portfolio for sat. In *SAT competition*, 2004.

[14] E. Nudelman, K. Leyton-Brown, A. Devkar, H. Hoos, and Y. Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In *Conf. on Princ. and Prac. of Const. Prog.*, 2004.

[15] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2002.

[16] H. Zhang. SATO: an efficient propositional prover. In *Conf. on Auto. Deduc.*, pages 272–275, 1997.

[17] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Termination Criteria for Bounded Model Checking: Extensions and Comparison [1]

Mohammad Awedh [2] Fabio Somenzi [3]

University of Colorado at Boulder

Abstract

Increasing attention has been paid recently to criteria that allow one to conclude that a structure models a linear-time property from the knowledge that no counterexamples exist up to a certain length. These termination criteria effectively turn Bounded Model Checking into a full-fledged verification technique and sometimes result in considerable time savings. In [1] we presented a criterion based on the translation of the linear-time specification into a Büchi automaton. BMC can be terminated if no fair cycle is found up to a given length, and one can prove that no fair cycle exists beyond that length. The maximum length for which counterexamples are explicitly checked is called the *termination length*; it obviously depends on the model, the property, and the termination criterion. In this paper we improve the criterion of [1] by adding a check that often substantially reduces termination length. Our previous work employed translation to a *non-generalized* Büchi automaton. Though a well-known technique converts a *generalized* automaton into that form by composing it with a counter, it has the undesirable effect of considerably lengthening the cycles in the graph to be searched. We propose several alternatives to that approach and compare them experimentally. The translation to automata can be accomplished in more than one way, and in this paper we contrast two of them: one based on the algorithms of [18], and one based on the notion of *tight automaton* of [5]. The latter yields shorter counterexamples, but the former often leads to earlier termination. In addition, it can help in identifying safety properties, for which termination checks are much more efficient than for the general case. We finally present results on comparing techniques based on cycle detection to the technique of [13], which converts liveness properties into safety properties by augmentation of the model.

Key words: bounded model checking, Büchi automata, safety and liveness properties, termination conditions.

[1] This work was supported in part by SRC contract 2004-TJ-920.

[2] Email: Awedh@Colorado.EDU

[3] Email: Fabio@Colorado.EDU

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Bounded Model Checking (BMC, [3]) is a model checking approach for linear time properties typically expressed in Linear Time Logic (LTL). BMC reduces the search for a counterexample to an LTL property to propositional satisfiability (SAT). Given a Kripke structure $\mathcal{M}$, an LTL formula ψ, and a bound k, BMC tries to refute $\mathcal{M} \models \mathsf{A}\,\psi$ by proving the existence of a witness of length k to $\neg\psi$. That is, BMC tries to find a witness to $\mathcal{M} \models_k \mathsf{E}\,\neg\psi$. The k-bounded witness to $\mathcal{M} \models_k \mathsf{E}\,\neg\psi$ is a path in $\mathcal{M}$ with at most k states. It can be either a finite prefix of a path for a safety property or a looping path (a k-loop) in the general case.

The standard technique to check an LTL property [22,11] constructs a Büchi automaton that accepts all the counterexamples to the LTL formula, and then checks the composition of the property automaton and the original model for language emptiness. The size of the automaton is exponential in the length of the LTL property, and this technique is in PSPACE [17]. Language emptiness is often checked by a BDD-based fixpoint computation.

BMC is known to be a complementary method to the BDD-based LTL model checking: Many problems that are hard for the BDD-based method can be solved easily by BMC [7]. However, it is hard to predict in advance the cases where BMC is more efficient than the BDD-based method [19].

The original Bounded Model Checking [3], although complete in theory, is limited in practice to falsification of LTL properties. BMC can prove that an LTL property ψ passes on a model $\mathcal{M}$ only if a bound, κ, is known such that, if no counterexample of length up to κ is found ($\mathcal{M} \not\models_\kappa \mathsf{E}\,\neg\psi$,) then $\mathcal{M} \models \mathsf{A}\,\psi$. Several methods exist to compute a suitable κ, all of them depending on $\mathcal{M}$, ψ, and the BMC encoding scheme. Some methods are straightforward, but are usually poor. (For an invariant property, one could use an upper bound on the number of reachable states as κ.) The optimum value of κ, however, is usualy very expensive to obtain: Finding it is at least as hard as checking whether $M \models \mathsf{A}\,\psi$ [6].

Several practical approaches that over-approximate the value of κ are based on the recurrence diameter/radius of the model [3]. In [15], the authors use the forward and backward recurrence radii to prove invariant properties. Their approach is based on the observation that if a counterexample to an invariant exists, then there is a simple path from an initial state to a failure state that goes through no other initial or failure state. An invariant holds if all states of all paths of length k starting from the initial states satisfy the invariant, and moreover, there is no simple path of length $k+1$ starting at an initial state or leading to a failure state, and not going through any other initial or failure states. In this approach the forward and backward recurrence radii can be found by solving a sequence of SAT instances rather than QBF instances. Hence, they are easier to compute. However, the bound based on recurrence radii is not as tight as the one based on the radii.

The approach of [12] does not explicitly compute the backward radius, but it only examines counterexamples of length up to it to prove termination in invariant checking. For a given value of k, the algorithm iterates over approximations of

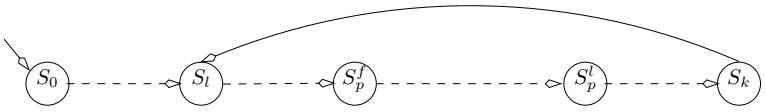

Fig. 1. Path to a fair cycle

the states that are reachable from the initial states, R. The algorithm starts by setting R to be the set of initial states. At each iteration of the algorithm, an over-approximation of the states that are reachable from R in one step is added to R. If no more states are added to R and R does not intersect the target states, the safety property holds in $\mathcal{M}$. It can be shown that k is less than or equal to the backward radius of $\mathcal{M}$.

For safety properties one can compose a suitable automaton with the model to be verifi ed. In this way, model checking is reduced to reachability and the termination criteria for invariants can be applied. General LTL properties, on the other hand, require the ability to prove language emptines (i.e., detect cycles), and therefore call for different approaches to check termination.

One such approach [2] involves a translation that reduces the check for simple liveness to the check for an invariant on an augmented model. The translation adds two components to the model: one for loop detection and the other to monitor the property. This addition doubles the state variables and correspondingly increases the number of reachable states and the length of the longest shortest/simple paths.

A tight bound on proving simple liveness properties is proved in [13]; the upper bound for checking a simple liveness property in BMC is ($\neg p$-predicated radius + $\neg p$-predicated diameter). The $\neg p$-predicated radius (diameter) is the radius (diameter) of the model after deleting all states in which p holds.

In [1], we presented a termination criterion based on the observation illustrated in Figure 1: A counterexample to $\mathsf{A}\,\mathsf{F}\,\mathsf{G}\,\neg p$ consists of two paths: A simple path form an initial state S_0 to a state that satisfi es p (S_p^l in Fig. 1), and a simple path leading back to a state that satisfi es p (S_p^f in Fig. 1) along which all other states satisfy $\neg p$. If no counterexample is found up to the sum of the bounds on the above simple paths, then $\mathsf{A}\,\mathsf{F}\,\mathsf{G}\,\neg p$ holds. Our termination criterion takes into account the position of the states that satisfy the fairness constraint p and does not augment the model.

After the preliminaries of Section 2, in Section 3 we improve on the criterion of [1] by introducing an additional check that often reduces the length of counterexamples that must be explicitly examined. In Section 4 we study the effect of the translation from LTL to automata on the length of counterexamples and termination. In Section 5 we extend the technique to deal directly with generalized B¨uchi automata. The results of our experiments are reported in Section 6, and conclusions are offered in Section 7.

2 Preliminaries

LTL Model Checking is the problem of checking whether a specification, expressed by an LTL formula ψ, holds on all paths of a model $\mathcal{M}$, $\mathcal{M} \models \mathsf{A}\,\psi$. The *LTL formulae* over atomic propositions AP are defined as follows

- Atomic propositions, true, and false are LTL formulae.
- If ψ and ϕ are LTL formulae, then so are $\neg\psi$, $\psi \wedge \phi$, $\psi \vee \phi$, $\mathsf{X}\,\psi$, and $\psi\,\mathsf{U}\,\phi$.

An LTL formula that does not contain the temporal operators (X and U) is *propositional*. We write $\psi\,\mathsf{R}\,\phi$ for $\neg(\neg\psi\,\mathsf{U}\,\neg\phi)$, $\mathsf{F}\,\psi$ for $\mathsf{true}\,\mathsf{U}\,\psi$, and $\mathsf{G}\,\psi$ for $\mathsf{false}\,\mathsf{R}\,\psi$.

In LTL model checking, the behavior of the model is usually described by a *Kripke structure*. A Kripke structure $\mathcal{K} = \langle S, \delta, I, L\rangle$ consists of a finite set of states S whose connections are described by the transition relation $\delta \subseteq S \times S$. If $(s,t) \in \delta$, then there is a transition form state s to state t in $\mathcal{K}$. The transition relation δ is total, i.e., for every state $s \in S$ there is a state $t \in S$ such that $(s,t) \in \delta$. $I \subseteq S$ is the set of initial states of the model. The labeling function $L : S \to 2^{AP}$ indicates what atomic propositions hold at each state. We write $\delta(s,t)$ for $(s,t) \in \delta$; that is, we regard δ as a predicate. Likewise, we write $I(s)$ to indicate that s is an initial state, and, for $p \in AP$, $p(s)$ to indicate that $p \in L(s)$.

A path π in $\mathcal{K}$, whether finite or infinite, is a non-empty sequence $(s_0, s_1, \ldots)$ of states in $\mathcal{K}$ such that $\delta(s_i, s_{i+1})$ for all $0 \leq i < |\pi|$, where $|\pi|$ is the path length. We let $\pi(i) = s_i$ be the i-th state of π, $\pi_i = (s_0, \ldots, s_i)$ be the prefix of π and $\pi^i = (s_i, s_{i+1}, \ldots)$ be the suffix of π.

LTL formulae are interpreted over infinite paths. An atomic proposition p holds along a path $\pi = (s_0, s_1, \ldots)$ if $p(s_0)$ holds. Satisfaction for true, false, and the Boolean connectives is defined in the obvious way; $\pi \models \mathsf{X}\,f$ iff $\pi^1 \models f$, where $\pi^i = (s_i, s_{i+1}, \ldots)$; and $\pi \models f\,\mathsf{U}\,g$ iff there exists $i \geq 0$ such that $\pi^i \models g$, and for $j < i$, $\pi^j \models f$.

The *diameter* d of $\mathcal{K}$ is the length of the longest shortest path in $\mathcal{K}$. That is, the diameter is the maximum finite distance between two states. The (forward) *radius* r of $\mathcal{K}$ is the maximum finite distance in $\mathcal{K}$ of a state from the closest states in I. In other words, the radius is the maximum number of forward transitions needed to reach a state reachable from the intitial states. The *backward radius* is the maximum number of backward transitions from a given set of states needed to reach a state backward-reachable from those states.

A simple path between any two states s and t in $\mathcal{K}$ is a cycle free path. The *recurrence diameter* rd of $\mathcal{K}$ is the longest simple path in $\mathcal{K}$. The *recurrence radius* rr of $\mathcal{K}$ is the longest simple path in $\mathcal{K}$ that starts form a state in I.

A finite sequence of states can represent an infinite path if it contains a loop. A path $(s_0, \ldots, s_k)$ is a (k,l)-loop path if there is a transition from state s_k to state s_l for some $l \leq k$. A path is a k-loop path if it is a (k,l)-loop path for some $l \leq k$.

A *safety* property is such that every counterexample to it has a finite prefix that, however extended to an infinite path, yields a counterexample. Sistla [16] provides a syntactic characterization of LTL safety formulae: Every propositional formula

is a safety formula, and if f and g are safety formulae, then so are $f \vee g$, $f \wedge g$, $\mathsf{X}\, f$, $\mathsf{G}\, f$, and $f\, \mathsf{R}\, g$. Not all safety properties are captured by this definition.

Though in principle a counterexample to a linear-time property is always an infinite sequence of states, for safety properties it is sufficient and customary to present an initialized simple path that leads to a *bad state*—one from which all extensions to infinite paths result in counterexamples.

A *Büchi automaton* over alphabet Σ is a quadruple $\mathcal{A} = \langle Q, \Delta, q_0, F \rangle$, where Q is the finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states (or fair set). A *run* of $\mathcal{A}$ over an infinite sequence $w = (w_0, w_1, \ldots) \in \Sigma^\omega$ is an infinite sequence $\rho = (\rho_0, \rho_1, \ldots)$ over Q, such that $\rho_0 = q_0$, and for all $i \geq 0$, $(\rho_i, w_i, \rho_{i+1}) \in \Delta$. A run ρ is *accepting* if there exists $q_j \in F$ that appears infinitely often in ρ. Every LTL formula ψ can be translated into a Büchi automaton $\mathcal{A}_\psi$ such that $\mathcal{A}_\psi$ accepts exactly the paths that satisfy ψ [9,18]. In the automata-based approach to LTL Model Checking [20], a Büchi automaton that accepts counterexamples to the LTL formula is constructed. Then, the existence of an initialized fair cycle in the composition of the model and that automaton indicates failure of the specification.

3 An Improved Criterion for Language Emptiness

Let $\mathcal{K} = \langle S, \delta, I, L \rangle$ be a Kripke structure, and let $p \in AP$ be an atomic proposition. Let $path_k$ ($simplePath_k$) be the predicate that is true if $(s_0, \ldots, s_k)$ is a (simple) path in $\mathcal{K}$. The method of [1] is based on the following result.

Theorem 3.1 *Let these predicates over $s_0, \ldots, s_k$ denote sets of paths in $\mathcal{K}$:*

$$\alpha_k = I(s_0) \wedge simplePath_k \wedge p(s_k) \tag{1a}$$

$$\beta_k = simplePath_{k+1} \wedge \neg p(s_k) \wedge p(s_{k+1}) \tag{1b}$$

$$\beta'_k = simplePath_{k+1} \wedge \bigwedge_{0 \leq i \leq k} \neg p(s_i) \wedge p(s_{k+1}) \tag{1b'}$$

$$[\![\mathcal{K}, \neg\, \mathsf{F}\, \mathsf{G}\, \neg p]\!]_k = I(s_0) \wedge path_k \wedge \bigvee_{0 \leq l \leq k} [\delta(s_k, s_l) \wedge \bigvee_{l \leq i \leq k} p(s_i)] \ . \tag{1c}$$

Let m be the least value of k for which β'_k is unsatisfiable, and n the least value of k for which $(\alpha_k \vee \beta_k)$ is unsatisfiable. Then, $[\![\mathcal{K}, \neg\, \mathsf{F}\, \mathsf{G}\, \neg p]\!]_k$ is unsatisfiable unless it is satisfiable for $k \leq n + m - 1$.

With reference to Fig. 1, n bounds the length of the path from S_0 to S_p^l, while m bounds the length of the path from S_p^l to S_p^f.

The value of m in Theorem 3.1 can be unnecessarily large in certain structures. As an example, consider Figure 2, in which some unreachable states of a structure are shown. If S_8 is the only state satisfying p, then $m = 8$ because of the simple path from S_0 to S_8. However, such a path cannot be used to close the loop (from S_p^l to S_p^f in Fig. 1) because its initial state does not satisfy p. The longest simple path whose first state satisfies p, such that p does not hold in any subsequent state

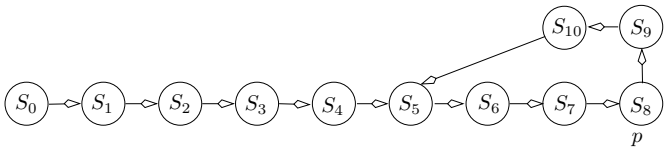

Fig. 2. Capturing the value of m

has length 5; it is therefore sufficient to take $m = 5$. To capture this observation, we add (1b″) to Theorem 3.1.

$$\beta''_k = simplePath_{k+1} \wedge p(s_0) \wedge \bigwedge_{1 \le i \le k+1} \neg p(s_i) \qquad \text{(1b″)}$$

Predicate (1b″) does not replace (1b′): If we move p in Figure 2 from state S_8 to state S_2, β'_k will capture the smallest value of m. In general, if the distance between p-states in the loops is shorter than the stem paths leading to them, β'' will be more effective than β'; conversely, when the p states appear only on the stem paths, β' will often give the smaller value of m. Hence, we let m be the minimum value of k for which either β'_k or β''_k becomes unsatisfiable.

Termination criteria benefit in general from knowledge of what states are unreachable. Performing full reachability analysis would make subsequent BMC redundant and is often too costly. However, reachability analysis of the Büchi automaton is usually quite cheap and identifies a subset of the unreachable states of the composite model. This subset can later be used as a constraint on the paths examined to decide termination by requesting that no state in those paths belong to the unreachable subset.

4 Tight Büchi Automata

In [14], the authors show that the construction of [5] yields a Büchi automaton that is *tight*, and therefore guarantees that shortest counterexamples in the composition of automaton and model will map onto shortest counterexamples in the original model. This property is not shared by the Büchi automata produced by the translation of [18] which we used in [1] to check for termination. For that reason, the algorithm proposed in [1] used the automaton approach only for termination check; for counterexample detection it used the BMC encoding of [3].

It is then natural to ask whether tight automata would benefit the scheme of [1], since they would allow one to unify the models used for proof and falsification of a property. We shall show, however, in Section 6 that most of the times the use of tight automata increases the termination length, often substantially.

One reason for this increase in termination length, which comes from increases of both m and n in Theorem 3.1, is to be found in the sharp increase in the number of states and transitions that the use of tight automata causes. In addition, a tight automaton has exactly one acceptance condition for each until operator in the LTL formula. By contrast, approaches like the one of [18] often reduce the number of acceptance conditions. A final important reason is related to the notion of automa-

ton strength [4]. Tight automata are almost always strong, while the automata of [18] are mostly weak and terminal. It can be shown that for terminal automata $m = 0$ in Theorem 3.1. For weak automata, this property does not hold, but experiments show that $m = 0$ most of the time. (In both cases, it is the β'' predicate that is responsible for these low values.)

Terminal automata accept co-safety languages. Since we detect properties that are syntactically safety properties, we apply the termination criterion for invariants to most safety properties even without resort to the strength of the automaton. However, the occasional safety property will fail the syntactic check, and, more importantly, properties producing weak automata are common, and are handled by our language emptiness criterion. A special criterion for weak properties can be devised; however, its impact on performance is small.

Tight automata are also to be considered as replacement of the traditional BMC encoding of [3] for counterexample detection. In fact, we have found that the encoding of [8,10], which are closely related to the use of tight automata, often improve the speed of counterexample search.

5 Checking Multiple Fairness Conditions

The approach of Theorem 3.1 for checking language emptiness only considers Büchi automata with one fair set. Büchi automata with multiple fair sets are known as *generalized* Büchi automata. The acceptance condition of a generalized Büchi automaton is a set of fair sets $\mathcal{F} \subseteq 2^Q$. A run of a generalized Büchi automaton is accepting if some state p_i of each of the sets $F_i \in \mathcal{F}$ appears infinitely often.

One solution to handle multiple fair sets $\mathcal{F} = \{F_1, \ldots, F_r\}$ is to convert a generalized Büchi automaton $\mathcal{G}$ to an equivalent Büchi automaton $\mathcal{A}$. The standard construction composes a counter with $\mathcal{G}$. The counter has $r = |\mathcal{F}|$ states, is initialized to 1, and is incremented from i to $i + 1$ when a state in F_i is visited. It is reset from r to 1 when a state in F_r is visited. A fair state in $\mathcal{A}$ is a state in which $\mathcal{G}$ satisfies F_1 and the counter's value is 1.

The conversion expands the size of the automaton by a factor related to the number of fair sets. The effectiveness of our termination criterion depends on the order at which the fair sets in $\mathcal{F}$ are visited and in general is reduced as the number of fair sets increases because the fair cycles of the non-generalized automaton tend to grow longer and longer.

To counter this effect, instead of using a counter, we add a state variable x_j for each fair set F_j which keeps track of when a state in F_j is visited.

$$\begin{aligned} x_j(0) &= p_j(0) \\ x_j(t) &= p_j(t) \vee x_j(t-1) \wedge \neg x(t-1) \ . \end{aligned}$$

A fair state in the Büchi automaton is one thata satisfies $x = \bigwedge_{j=1}^{r} x_j$. We can then check the conditions of Theorem 3.1 on x. We call this method the *Flag* method.

Another solution is to extend Theorem 3.1 to handle multiple fair sets. One way

Table 1
Using β''

Model	β'			β''			Both		
	m	n	$T(s)$	m	n	$T(s)$	m	n	$T(s)$
Arbiter	7	-	128.7	6	-	**117.8**	6	-	130.3
Abp	2	-	215.2	2	-	**214.7**	2	-	270.3
D4	11	11	37.37	0	11	29.99	0	11	**28.33**
Fabric1	8	8	2.71	0	8	1.28	0	8	**1.26**
Fabric2	8	8	32.15	0	8	20.69	0	8	**20.33**
Feistel	2	9	2.77	0	9	2.63	0	8	**2.59**
FPMult	3	3	3.26	0	3	2.67	0	3	**2.49**
Huffman1	1	1	21.83	0	1	17.29	0	1	**17.01**
Huffman2	3	3	54.25	0	3	**42.56**	0	3	43.01
Huffman3	10	10	100.57	0	10	14.24	0	10	**13.68**
Miim1	3	3	0.17	0	3	**0.14**	0	3	0.17
Miim2	-	-	109.59	0	-	**108.37**	0	-	108.97
PPC60X_bus	-	-	TO	0	-	**1002.2**	0	-	1010.93
Smult	-	-	8.95	0	-	8.77	0	-	**8.64**
TicTacToe1	11	11	476.22	0	11	**261.75**	0	11	262.02
TicTacToe2	1	1	518.78	0	1	**423.68**	0	1	424.42
Tlc	0	15	**2.07**	0	15	2.13	0	15	2.12
Vsa16	-	-	**283.03**	0	-	296.61	0	-	294.33

to achieve that is to check language emptiness for the fairness condition $\bigcup_{F\in\mathcal{F}} F$. This approach, which we call the *Or* approach, will in general be conservative in estimating the termination length, but does not require any increase in the number of states of the automaton. Besides, taking the union of the fair sets may decrease the distance between fair states along the loops and hence reduce the value of m.

Yet another way to deal with multiple fairness constraints is to apply Theorem 3.1 to each fairness condition in turn. This *One* method also produces conservative values for m and n, but compared to the *Or* approach, will be more effective when one fairness constraint cannot be satisfi ed in isolation.

6 Experimental Results

The results presented in this section are for models that are from industry, and from the Texas-97 and VIS Verifi cation Benchmark sets [21]. For each model, we count each LTL property as a separate experiment. For all experiments, we set the maximum value of k to 30 and we check for termination at each step. The

Table 2
Using Automaton Reachability analysis

Model	*No*	*Yes*	Model	*No*	*Yes*
CacheCo	29.28	**21.68**	D4	29.31	**26.34**
Ethernet	109.53	**96.7**	Heap	513.12	**421.92**
HourGlass	504.55	**478.87**	Needham	1540.1	**1369.8**
PI_BUS	377.75	**366.92**	ProdCell	**85.47**	118.44
ReqAck	**101.85**	113.27	TwoFifo	95.75	**87.98**
TwoQ	235.37	**225.58**	VsaR1	**353.99**	366.61

Table 3
Comparing tight and non-tight automata

Model	St	Strength	Non-tight			Tight		
			m	n	$T(s)$	m	n	$T(s)$
Coherence	U	strong	17	-	TO	24	-	TO
Ifetch1	P	weak	0	2	**0.31**	-	-	25.77
Ifetch2	P	terminal	0	3	**0.15**	-	-	26.78
FPMult1	P	terminal	0	3	**5.77**	2	-	25.7
FPMult2	P	terminal	0	3	**2.79**	2	-	30.79
Microwave	P	weak	0	0	**0.1**	0	8	0.3
Pathfinder	P	weak	0	0	**0.1**	0	-	22.00
PI_BUS	P	weak	0	1	**0.57**	0	-	945.33
ReqAck	U	weak	0	-	**20.15**	-	-	27.22
s1269-1	P	weak	0	8	**0.22**	0	9	0.39
s1269-2	P	weak	0	8	**0.20**	0	16	1.1
s1269-3	P	terminal	0	1	**0.09**	0	-	81.39
s1423	P	terminal	0	4	0.16	3	4	**0.13**
UsbPhy	P	weak	0	-	143.9	1	-	**88.55**

experiments were run on an IBM IntelliStation with a 1.7 GHz Pentium IV CPU and 2 GB of RAM running Linux. The datasize limit was set to 1.5 GB.

The first column in every table is the name of the model. The columns labeled m and n in each table give the values of m and n in Theorem 3.1 respectively; if an entry in these columns is a dash, it indicates that no value is captured. The columns labeled T give the runtimes in seconds; boldface is used to highlight best runtimes; a TO in this column indicates a time greater than 1800 s. CPU times in all tables are for both counterexample detection and termination checks.

In Table 1 one sees that applying β'' reduces the value of m. For model *D4*, for instance, the value of m that is captured by β' is 11 while the value that is captured

Table 4
Safety properties

Model	General LTL			Safety		
	St	tl	$T(s)$	St	tl	$T(s)$
Fabric	P	8	20.42	P	8	**17.51**
Huffman1	P	1	17.36	P	1	**11.56**
Huffman2	P	3	43.2	P	3	**31.82**
Huffman3	P	11	17.05	P	10	**9.85**
Lock	U	-	750.45	U	-	**129.34**
Rrobin	U	-	TO	U	-	**160.19**
VsaR	P	5	361.83	P	5	**285.69**

by β'' is 0. Hence, the search for counterexample will stop after k=10 if β'' is used compare to k=21 if only β' is used. In many cases the overhead for checking β'' in addition to β' is well within the noise margin.

Table 2 compares the use of reachability analysis of the automaton when searching for the values of m and n (columns labeled *Yes*) to its omission (columns labeled *No*). Reachability analysis of the automaton usually reduces runtime, but it does not help in reducing the values of m and n.

Table 3 compares tight to non-tight *Büchi* automata when searching for a simple path. The column labeled St in this table indicates whether each property passes (*P*), or remains undecided (*U*). The column labeled $Strength$ is the automaton strength. From this table, we can conclude that using tight automata increases the termination length. The termination length for the model *s1423* increases from 3 to 6 when using a tight automaton. In model *Coherence*, the value of m is increased from 17 to 24 when applying a tight automaton. In model *Ifetch*, using a tight automaton does not even capture the value of m for a given value of k.

Table 4 illustrates the importance of a dedicated criterion for safety properties. All properties in this table are passing properties. The column labeled St has the same meaning as in Table 3; the column labeled tl, when present, reports the termination length.

Tables 5 and 6 show the results of applying different methods when handling multiple fairness conditions. The columns headings identify one of the *Or*, *One*, *Flag*, *Counter*, and *Trans* methods. The last one is the method that applies the translation of liveness into safety [2]. Safety checking is performed using both Bounded Model Checking algorithm (*bmc*) and the BDD-based LTL model checking algorithm in VIS (*ltl*).

Table 5 compares the first three methods. Table 6 compares all methods on a smaller set of examples, because for *Counter* and *Trans* we manually translated the examples. The upper part of Table 6 shows the results of applying the methods *Or*, *One*, and *Flag*. The lower part shows the results of applying the *Counter* and *Translation* methods.

Table 5
Comparison of Or, One, and Flag

Model	# fair	OR			One			Flag		
		m	n	$T(s)$	m	n	$T(s)$	m	n	$T(s)$
Am2910-1	1	0	1	6.02	0	1	6.62	0	1	**5.12**
Am2910-2	1	0	-	**43.24**	0	-	46.35	0	-	45.35
Arbiter	3	6	-	298.77	0	-	105.95	-	-	**32.51**
Chameleon	5	0	-	TO	0	1	0.1	0	0	**0.02**
Cups	7	2	-	TO	0	1	0.19	0	0	**0.04**
D12	6	0	-	22.33	0	3	**0.71**	6	7	1.17
D16	4	-	-	TO	0	1	**1.49**	-	-	TO
Dcnew	2	2	-	TO	0	0	**0.7**	0	-	TO
Nim	2	1	-	86.21	0	1	**0.14**	1	-	49.35
NulMdm	2	-	-	133.11	0	1	196.67	-	-	**107.29**
Philo	11	6	-	TO	0	-	TO	0	0	**0.03**
PnPong1	3	1	9	**1.36**	0	6	1.47	3	11	5.3
PnPong2	3	1	5	0.14	0	1	**0.12**	1	4	0.15
Pong-1	3	-	-	99.32	0	-	287.1	1	5	**0.35**
Pong-2	3	-	-	336.25	0	-	438.25	-	-	**318.42**
ReqAch	2	1	-	3.79	0	-	5.19	2	-	**3.12**
Rether	5	0	-	59.15	0	-	379.02	0	0	**0.07**
Rether	5	0	-	29.16	0	-	392.42	0	0	**0.09**
Short	3	0	2	0	0	0	0	0	0	0
Soap	2	-	-	TO	0	1	**0.35**	-	-	1523.43
Tlc-1	3	0	9	**0.37**	3	14	21.15	2	18	35
Tlc-2	2	0	15	**2.11**	0	15	6.9	4	18	13.95
Vendng1	2	14	-	TO	0	-	TO	13	-	**1041.63**
Vendng2	2	2	2	0.14	0	2	**0.11**	-	-	TO
Vendng3	2	2	19	1240.29	0	2	**0.18**	13	-	TO

In Table 5, 20 out of 25 properties are decided passed by at least one method within the given limit of time and value of k. Both methods *One* and *Flag* are the fastest in 8 experiments; *Or* is the fastest in only 3 experiments. No methods dominates the other in reducing the length of m and n. In model *PnPong1*, the *One* method captures the smallest values of m and n. While in model *Tlc-1*, the *Or* method captures the smallest values of m and n. The *Flag* method proves properties pass in zero value of k in 6 experiments.

Table 6 shows that, the values of m and n that are found by the *Flag* method

Table 6
Comparison of Or, One, Flag, Counter, and Trans.

Model	# fair	Or			One			Flag		
		m	n	$T(s)$	m	n	$T(s)$	m	n	$T(s)$
Crd1	5	0	14	8.5	0	14	25.05	7	18	212.64
Crd2	5	0	26	720.51	0	1	0.05	7	-	659.39
$\mu 1$	2	4	6	0.15	0	6	0.64	0	9	0.39
$\mu 2$	2	0	0	0	0	1	0.01	5	6	0.11
$\mu 3$	2	3	4	0.3	0	3	0.12	5	7	1.84
Pong	3	-	-	101.86	0	-	299.74	1	5	0.35
Short	3	0	2	0	0	0	0	0	0	0

Model	Counter			Trans(bmc)		Trans(ltl)
	m	n	$T(s)$	k	$T(s)$	$T(s)$
Crd1	18	19	66.73	22	493.25	0.21
Crd2	22	-	128.73	30	890.56	0.15
$\mu 1$	9	12	1.6	13	5.32	0.05
$\mu 2$	7	9	0.22	18	31.56	0.05
$\mu 3$	6	11	1.91	19	92.07	0.09
Pong	-	-	36.88	30	48.71	TO
Short	4	4	0.02	10	0.57	0.01

are smaller than those found by the *Counter* method. This is because the *Counter* method depends on the order in which the fair sets are visited, while the *Flag* method does not.

The *Trans* method blows up the state space and increases the diameter and radius of the model. In almost all the experiments in Tables 6, the *Trans* method, using *bmc*, is the slowest one. In almost all these experiments, *ltl* is very fast. We have not tried large models yet with the *Tran* and *Counter* methods because they required considerable manual work to convert them, but we expect that for larger examples *bmc* will prove faster more often.

7 Conclusions

We have presented an improved criterion for termination in Bounded Model Checking, which signifi cantly reduces termination length. An improvement to our termination check could be restricting the search using the *Flag* method to paths that end in a state that satisfi es at least one fairness constraint. We are currently evaluating this improvement.

We have also shown that the use of the reachability analysis of the property

automaton speeds up the termination check, but does not reduce the termination length. Even though tight automata find shortest k-loop counterexamples, they increase the termination length.

We have presented different methods for checking multiple fairness conditions when checking language emptiness using BMC. The *Flag* and *One* methods are the best among them. Both *Flag* and *Counter* methods are based on recording the visiting of fair states. However, the performance of the *Counter* method depends on the order of visit of the fair sets. The *Flag* method helps limiting the search for a simple path to the ones satisfy all fairness constraints. Hence, it helps finding small values for m and n, but it may increases the searching time.

The efficiency of the *One* method also depends on the order in which fair sets are checked; in practice, we found that it is more efficient to check the fair states that come from the property automaton before those supplied with the model.

References

[1] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 96–108. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[2] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2), July 2002. Formal Methods for Industrial Critical Systems (FMICS'02).

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[4] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633.

[5] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 415–427. Springer-Verlag, Berlin, 1994. LNCS 818.

[6] E. Clarke, D. Kröning, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 85–96, Venice, Italy, Jan. 2004. Springer. LNCS 2937.

[7] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In G. Berry, H. Comon, and A. Finkel, editors, *Thirteenth Conference on Computer Aided Verification (CAV'01)*, pages 436–453. Springer-Verlag, Berlin, July 2001. LNCS 2102.

[8] A. Frisch, D. Sheridan, and T. Walsh. A fixpoint based encoding for bounded model checking. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 238–255. Springer-Verlag, Nov. 2002. LNCS 2517.

[9] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[10] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. In *Formal Methods in Computer Aided Design*, pages 186–200, Austin, TX, Nov. 2004. Springer. LNCS 3312.

[11] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Jan. 1985.

[12] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[13] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Software Tools for Technology Transfer*, 5(2–3):185–204, Mar. 2004.

[14] V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, pages 493–509, Edinburgh, UK, Apr. 2005.

[15] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 108–125. Springer-Verlag, Nov. 2000. LNCS 1954.

[16] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects in Computing*, 6:495–511, 1994.

[17] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 3(32):733–749, 1985.

[18] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[19] O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, Jan. 2004.

[20] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[21] URL: http://vlsi.colorado.edu/∼vis.

[22] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

Supporting SAT based BMC on Finite Path Models

Daniel Geist, Mark Ginzburg, Yoad Lustig
Ishai Rabinovitz, Ohad Shacham and Rachel Tzoref
geist@il.ibm.com

IBM, Haifa Research Labs

Abstract

The standard translation of a Bounded Model Checking (BMC) instance into a satisfiability problem, (a.k.a SAT), might produce misleading results in the case when the model under verification contains finite paths. Models with finite paths might be produced unknowingly when using modern verification languages such as PSL-Sugar [1]. Specifically, the use of language constructs such as *restrict*, *assume* etc. might lead to such models. Thus the user may receive misleading results from SAT based tools.

In this paper we describe in what circumstances the finite path problem occurs and present an improved translation of the BMC problem into a SAT instance. The new translation does not suffer from the discussed shortcoming. Our translation is only slightly longer then the usual one introducing one extra Boolean variable in the model.

We also show that this translation may improve the SAT solver runtime even for models without finite paths.

Key words: BMC, PSL, Finite paths

1 Introduction

Since its introduction in the seminal paper [5], SAT-based Bounded Model Checking (BMC) has become an important tool in the verification engineer toolbox. However, traditional translation of a Bounded Model Checking instance into a satisfiability problem (a.k.a SAT) is not perfect. In particular it might produce misleading results when the model under verification contains a finite path that violates the specification.

In this paper we describe this problem and in what circumstances it might occur. Models with finite paths might be produced unknowingly when using modern verification languages such as PSL-Sugar [1]. Specifically, the use of language constructs such as *restrict*, *assume* etc. might lead to such models.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Thus the user may receive misleading results from SAT based tools. We also present an improved translation of the BMC problem into a SAT instance that does not suffer from the discussed shortcoming. Our translation is only slightly longer then the usual one introducing one extra Boolean variable in the model.

Our improved translation not only fixes the problem when finite paths exists, it may also improve the performance of the SAT solver even for models with no finite paths.

The rest of the paper is divided as follows: Section 2 overviews the standard translation of the BMC problem to a SAT problem. Section 3 presents examples of models with finite paths. Section 4 presents our solution. Section 5 details run time results that show that the new translation can assist in runtime and Section 6 presents our conclusions.

2 Translating a BMC Problem to a SAT Problem

The usual way a BMC problem is translated to a SAT instance is quite simple. Before describing the translation itself, we introduce several notations:

2.1 Notations

Denote by s_i a vector of propositional variables encoding the state of the model in cycle i. Denote by $INIT(o)$ the propositional formula translation the initial set, i.e. $INIT(s_0)$ encodes "The state s_0 is in the set of initial states". Denote by $TR(o, o)$ the function encoding the transition relation, i.e. $TR(s_{i-1}, s_i)$ encodes "There is a transition from state s_{i-1} to state s_i". A *computation path* is a sequence of states $s_0, s_1, \ldots, s_n$ such that s_0 is in the initial set and for each two consecutive states s_{i-1}, s_i there is a transition from state s_{i-1} to state s_i (i.e. $TR(s_{i-1}, s_i) = 1$).

Finally we introduce the notion of a bad state. Recall that BMC can be applied to formulas of the form *always p*. Since the specification is of the type *always p* [1] , the specification can be seen as an invariant, a bad state is a state that violates the specification invariant. For example, if the specification is *always p*, a bad state will be any state not satisfying p. Denote by $BAD(o)$ the formula translation of the bad states, i.e., $BAD(s_i)$ encodes "The state s_i is a bad state".

A *bug* is a bad state that is reachable by a computation path.

[1] In fact, many other formulas can be transformed to formulas of the type always p, possibly adding to small monitoring automata to the model. SAT based BMC methods can then handle all safety formulas [3], and even liveness formulas (although in the latter case, the complexity price is significant).

2.2 The translation

The standard translation of BMC into a satisfiability problem is to find a satisfying assignment to the following equation:

$$(1) \qquad INIT(s_0) \wedge (\bigwedge_{i=1}^{k} TR(s_{i-1}, s_i)) \wedge (\bigvee_{i=0}^{k} BAD(s_i))$$

A bug can be reached within k cycles iff the traditional BMC formula is satisfiable. Furthermore, a satisfying assignment to the BMC formula can be translated in a straightforward manner to a counter-example trace leading to the bug.

In practice, the verification engineer fixes the length parameter k, for example $k = 10$. The tool produces the formula and feeds it into a SAT solver. If the formula is satisfiable then a bug is found, otherwise, the result is seen as "k-passed", i.e., the model does not contain a bug in paths up to length k. Some modern verification tools (such as IBM's RuleBase [4]) also provide automatic modes in which the bound k is automatically increased until a bug is found or the system runs out of resources.

```
VAR xx : 0..7;

ASSIGN
    init (xx)  := 0;
    next(xx)  := case
        xx  =  7 :  0;
        else      :  {xx,  xx + 1};
    esac;

assume always((xx != 7)  |  (next(xx)  =  2));
```

Fig. 1. Simple model with a finite path

3 Models with Finite Paths

Model checking is usually performed on infinite paths models [6]. A finite computation path may result from a computation ending in a state for which no other state satisfies the transition relation. For example, the model depicted in Figure 1 has finite paths: a finite path may occur if xx reaches 7. In that case, xx must both become 0 to satisfy the $next(xx)$ assignment, and become 2 to satisfy the *assume* statement. Clearly no value can satisfy both constraints therefore the path has no continuation and is finite.

In modern verification languages such as PSL-sugar, finite paths might occur from constructs such as *assume* and *restrict* statements, using next variables on the right hand side of an assignment statement, and from assume or restrict verification directives. We would like to stress that even though the example in Figure 1 is contrived, models with finite paths often occur in practice because engineers find assumptions to be a very useful convenience and that was our motivation in doing this work.

When dealing with a model that has finite computation paths, it is customary to define a bug as valid only if it is part of an infinite path. Still, it is also reasonable and desirable to define it as valid even it has no infinite extension for several reasons:

(i) In many cases the verification engineer is interested in bugs occurring on finite paths. In fact, the verification engineer often introduces verification directives (such as *restrict*) that might turn many or even all paths to finite ones. This is done in order to reduce the state space significantly and to "concentrate" the verification effort on parts of the state space that the verification engineer considers sensitive. For example, a verification engineer may use such directives in order to ignore paths with known bugs, so the verification engineer may turn an assert into an assume, however, any bug that occurs prior to the violation of the assume is relevant and should be reported.

(ii) Some of the main verification techniques used by modern tools, do not guarantee that a bug found is on an infinite path. The most notable are BDD-based On The Fly verification (e.g. IBMs Discovery engine inside RuleBase), and SAT based Bounded Model Checking. While both techniques can be adapted to ensure that a bug found can be reached on an infinite path, the cost (in terms of time and memory consumption) of this adaptation is significant, many times much bigger than the cost of finding the bug in the first place.

To see that SAT based BMC does not ensure that the bug found is on an infinite path, simply note that the BMC formula refers only to the first k cycles and there is no guarantee as to what happens after cycle k. There are SAT based algorithms that solve the unbounded MC problem but they do not scale to large designs as BMC.

Therefore, the approach today is to find bugs on finite as well as infinite path. While this approach does not follow the strict temporal logic definitions it enables the verification engineer to enjoy stronger tools (such as BDD based On The Fly algorithm, or SAT based BMC), as well as easier use of verification directives statements (such as *restrict*). See for example [7] that discusses the temporal logic semantics on finite paths and considers validity of bugs which appear only on finite paths.

Verification engineers that use BMC to find bugs expect BMC to be *consistent*: the expectation is that if BMC does not find a bug on a run with a bound of k , then it can not find bugs on any run with bound $k' < k$ .The consistency of BMC is a very important attribute, since it enables the verification engineer to increase k in increments grater than one without the risk of missing a bug.

The traditional BMC formula does not treat finite paths well enough. For example, the model in Figure 2 is deterministic, and therefore contains only one path. This single path is finite and of length 5 since at cycle 5 (when the first cycle being cycle 0) it holds that $xx = 5$. If $xx = 5$ the path has no continuation since no state can satisfy both the $next(xx)$ assignment statement and the *assume* statement.

$$
\begin{array}{l}
VAR\ xx : 0..7; \\
\\
ASSIGN \\
\quad init\ (xx)\ := 0; \\
\quad next(xx)\ := case \\
\qquad xx\ =\ 7 :\ 0; \\
\qquad else\ \ \ \ \ :\ xx + 1; \\
\quad esac; \\
\\
assume\ always((xx\ !=\ 5)\ \mid\ (next(xx)\ !=\ 6));
\end{array}
$$

Fig. 2. An example of a model with all paths being finite

3.1 The Bounded Paths Problem

The equation Eq. 1 encodes the following statement: "There is a computation path of length k, and somewhere on this path a bad state is encountered". Note however, that in a model in which all paths that violate the specification [2] are finite and are of length $k-1$ or less the formula is unsatisfiable although a bug might be encountered before cycle $k-1$. For example, look at the model in Figure 2 with the specification *always*$(xx < 3)$. The model violates the specification in the fifth cycle (the first cycle being cycle 0). However, if the verification engineer sets the bound k to 10, then Eq. 1 will be unsatisfiable because there are no paths of length 10. Thus the verification engineers seeing that the formula is unsatisfiable will classify the model as "10-passed", which is clearly wrong. Increasing the bound will not help and decreasing the bound is

[2] We neglect paths in which the bug is cycle larger than k, and cannot be found in this run of BMC in any case.

counter-intuitive for the engineer and it is impractical to ask him to consider it. So the end result will be a bug miss which is a very **severe** outcome. In other words, using SAT on a model which contains PSL assumptions absolutely requires handling this case.

By this example we can see that the presence of finite paths causes BMC to lose its consistently attribute, this is problematic even if the verification engineer is not interested in bugs that occur on finite paths, the reason is that the verification engineer cannot tell that a bug is on a finite path (the path may be long enough), and therefore can suffer from the following scenario:

(i) A BMC run with a bound of k passes since it ignores a violation in a finite k'-length path ($k' < k$).

(ii) As a result of a change in the design, the verification engineer runs BMC again and by chance uses a bound of $k'' < k'$. A bug is reported and the verification engineer assumes that this is a new bug, entered by the change in the design.

Such a scenario is obviously problematic.

4 Solution to finite path problem

A simple solution is to start with $k = 1$ and increment k by 1 on each iteration of the verification tool, then we are sure to catch the bug on the first k it appears. The problem with this solution is of course that it is extremely time consuming since many invocations of the verification tool are needed. A second solution is to encode into a propositional formula the statement "Either there is a path of length 1 leading to a bad state, or there is a path of length 2 leading to a bad state or... there is a path of length k leading to a bad state". This solution, while only invokes the SAT solver once, invokes it on a significantly longer formula (in fact of quadratic length compared to the original formula), which is extremely costly.

4.1 The Improved Translation

A better solution to the finite path problem can be achieved by changing slightly the traditional BMC formula. We introduce one extra Boolean variable to the model called **AlreadyFailed**. This variable records whether a bug has been hit on particular path. Mathematically:

- For an initial state s_0
  $AlreadyFailed(s_0) \leftrightarrow BAD(s_0)$
- For any other state s_i,
  $AlreadyFailed(s_i) \leftrightarrow (AlreadyFailed(s_{i-1}) \vee BAD(s_i))$

The BMC equation now becomes:

$$(2) \quad INIT(s_0) \wedge (\bigwedge_{i=1}^{k}(TR(s_{i-1}, s_i) \vee AlreadyFailed(s_{i-1}))) \wedge (\bigvee_{i=0}^{k} BAD(s_i))$$

Thus by adding this extra variable and changing the translation we are ensured of identifying a bug even if it is sitting on a finite path and the bound k we choose to submit to the model checker is greater than the length of that path.

In fact it is possible to simplify this translation in two ways:

(i) we can replace the definition of **AlreadyFailed** by
$AlreadyFailed(s_0) \rightarrow BAD(s_0)$
$AlreadyFailed(s_i) \rightarrow (AlreadyFailed(s_{i-1}) \vee BAD(s_i))$

(ii) we can replace the term $\bigvee_{i=0}^{k} BAD(s_i)$ in Eq. 2 with $AlreadyFailed(s_k)$.

However, these replacements may not necessarily provide a better runtime.

4.2 Implementation details

The new translation presented in Eq. 2 can in theory be applied always, even if the model under verification does not contains finite paths. In practice, It is not clear the effect of this translation to the performance of the SAT solver. It can slow the SAT solver since some optimizations cannot be performed when this translations is used, and since we added a new variable to the formula. On the other hand it may improve the SAT solver performance especially when the formula is satisfiable. The intuition is that the SAT solver attempts to find an assignment for all the variable replications until cycle k even if the bug is on cycle $k' < k$, In the new translation the SAT solver takes advantage of the *AlreadyFailed* variable replications to assign arbitrary values to variable replications belonging to cycles greater than k', and therefore may find the assignments faster.

For models with finite paths this is a necessity. Therefore the actual implementation has the following details:

The translation checks if the model contains PSL constructs that cause finite paths and chooses the translation according to that:

(i) When there can be finite paths there are several options:
  (a) It is recommended to use the new translation (Eq. 2).
  (b) The user can use the traditional translation, while advancing k by 1 each run. this way no bug is missed. However, this seems to be a very slow technique.
  (c) The user can force the use of the traditional translation (Eq. 1) using larger steps. However, the risk of missing a bug is taken after a conscientious decision.

(ii) When there can be no finite paths, the only issue in choosing the trans-

lation is the SAT solver performance. There are several options:

(a) Use the new translation (Eq. 2). This is recommended if it the user think that the SAT solver will find a satisfying assignment.

(b) Use the traditional translation. This is recommended when the user think that the SAT solver will not find a satisfying assignment.

(c) Use both translations and run two SAT solvers in concurrent, killing the slowest after the quickest gives a response. This is recommended for users that have the hardware resources.

The trace that is generated may contain states after the cycles that the bug occurred. In case the translation in Eq. 2 was used, those states can violate the constraints of $TR(o, o)$ and hence confuse the engineer. A postprocessing program has to remove those states from the trace before presenting it to the user.

5 Experimental results

As mentioned in Section 3 models with finite paths require the translation of Eq. 2 in order to identify an error so regardless of runtime improvement it is necessary to use the new translation. However, the new translation can also improve runtime. Table 1 details a comparison of the new and old translation in runs when the bound $k = 100$. This is a typical situation when a verification engineer starts verifying a new design. In this case, the engineer assumes, as is usually the case, that a bug exists in the first 100 cycles. All the SAT problems in Table 1 are satisfiable. They are taken from some proprietary industrial designs and from the IBM benchmarks [2]. Table 1 shows that in most of the cases, except for $D1$, the new translation significantly reduces the runtime. A reasonable explanation for such results is the fact that when a bug exists in a relatively small cycle then the new translation makes it much easier for the SAT solver to find a satisfying assignment for the formula variables in the higher cycles.

Table 2 details another realistic usage methodology. In practice, users run successive SAT in bound increments of 10 or 20 until a bug is found or a desirable cycle limit is reached. When the model contains finite paths the traditional translation can only be used with increments of 1. In order to use larger increments the new translation has to be used. All the models in the table contain finite paths. The first column in Table 2 specifies if the SAT problem is satisfiable or not. The second column specifies in which cycle the bug is found or the desirable limit was reached. The fourth column details the runtime using the old translation with 1 cycle increments. The fifth and last columns detail the runtime with the new translation using 10 and 20 cycle increments. In all of the examples we tried, incrementing the bound by 20 or by 10 was faster than incrementing by 1. Note that there is no reason to use the new translation with increments of 1 since its advantage is when the bug

is not on the last cycle.

	New trans. k=100	Old trans. k=100
batch_1_11	**140**	3201
D1	23	13
batch_29	**142**	192
batch_20	**826**	4039
batch_22	**3203**	13383
batch_18	**490**	3293
D2	**237**	250

Table 1
New translation Vs. Old translation. The runtime is displayed in seconds.

	SAT/UnSAT	No. Cyc.	Old Trans. inc. 1	New Trans. inc. 10	New Trans inc. 20
D1	SAT	40	18773	**948**	3221
D3	UnSAT	60	37199	3112	**2115**
D4	UnSAT	60	18818	884	**411**
D5	UnSAT	10	1176	**196**	—
D6	UnSAT	15	8265	**4787**	—
D7	UnSAT	100	14712	685	**394**
D8	SAT	31	991	556	**159**
D9	UnSAT	60	>145000	1916	**1763**
D10	UnSAT	100	40012	5622	**3059**

Table 2
comparing various increments of the new translation versus increments of 1 in the old translation. The runtime is displayed in seconds.

6 Conclusions

In this paper we presented a new encoding to SAT based BMC that enables BMC to effectively handle models with finite paths. This encoding is necessary for preserving the consistency of BMC on a model with finite paths

without serious performance degradation. In a certain cases, we have shown that the use of the new encoding increases the SAT solvers performance.

In the future, we plan to implement this new encoding in conjunction with incremental SAT based BMC [8][9].

References

[1] Property Specification Language: *Reference Manual*. Version 1.1, Accellera, June 2004.

[2] The IBM Formal Verification Benchmarks, http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html,2004.

[3] Ilan Beer, Shoham Ben-David, Avner Landver: *On-the-Fly Model Checking of RCTL Formulas*. CAV 1998.

[4] Shoham Ben-David, Cindy Eisner, Daniel Geist, Yaron Wolfsthal: *Model Checking at IBM*. Formal Methods in System Design 22(2): 101-108 (2003)

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: *Symbolic Model Checking without BDDs*. TACAS 1999.

[6] Edmund Clarke, Orna Grumberg, Doron Peled, *Model Checking*, MIT Press, 2000.

[7] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac and David Van Campenhout, *Reasoning with Temporal Logic on Truncated Paths*, CAV04.

[8] H. Jin, F. Somenzi. *An incremental algorithm to check satisfiability for bounded model checking*. BMC04

[9] Ofer Strichman: *Pruning Techniques for the SAT-Based Bounded Model Checking Problem*. CHARME 2001.

Computing Over-Approximations with Bounded Model Checking

Daniel Kroening [1]

Computer Science Department
ETH Zürich
Switzerland

Abstract

Bounded Model Checking (BMC) searches for counterexamples to a property ϕ with a bounded length k. If no such counterexample is found, k is increased. This process terminates when k exceeds the completeness threshold $\mathcal{CT}$ (i.e., k is sufficiently large to ensure that no counterexample exists) or when the SAT procedure exceeds its time or memory bounds. However, the completeness threshold is too large for most practical instances or too hard to compute.

Hardware designers often modify their designs for better verification and testing results. This paper presents an automated technique based on cut-point insertion to obtain an over-approximation of the model that 1) preserves safety properties and 2) has a $\mathcal{CT}$ which is small enough to actually prove ϕ using BMC. The algorithm uses proof-based abstraction refinement to remove spurious counterexamples.

1 Introduction

In the hardware industry, formal verification is well established. Introduced in 1981, *Model Checking* [10,12] is one of the most commonly used formal verification techniques in a commercial setting. However, it suffers from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [7].

This problem is partly addressed by a formal verification technique called *Bounded Model Checking* (BMC) [6], introduced by Biere and others. In BMC, the transition relation for a complex model M and its specification ϕ are jointly unwound up to a depth k to obtain a formula, which is then checked for satisfiability using a propositional SAT procedure such as Chaff [25]. In the case that ϕ is a safety property, the formula is satisfiable iff there exists a counterexample of length k, i.e., $M \not\models_k \phi$. If not so, k is increased to search for longer counterexamples. This

[1] Email: daniel.kroening@inf.ethz.ch

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

process terminates either if the SAT procedure exceeds its time or memory bounds, a counterexample is found, or k exceeds a *completeness threshold* $\mathcal{CT}$ [17]. In the later case, k is sufficiently large to ensure that no counterexample exists, and thus, we conclude $M \models \phi$. BMC has been used successfully to find subtle errors in very large industrial circuits [27,14].

The disadvantage of BMC is that it is typically only applicable for refutation; the best known completeness threshold for properties of type $\mathbf{G}p$ is the *reachability diameter* of M, i.e., the longest shortest path from any initial state to any reachable state in the state graph. In practice, the diameter is usually too hard to compute, and furthermore, is often exponential in the number of state variables in the model. The *recurrence diameter* [6] is an over-approximation of the reachability diameter. However, it is still difficult to compute and typically much larger than the reachability diameter.

Thus, in practice, the principal method for *proving* safety properties is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

In the hardware domain, the most commonly used abstraction technique is *localization reduction* [19,28,8]. The abstract model $\hat{M}$ is created from the given circuit by removing a large number of latches together with the logic required to compute their next state. The latches that are removed are called the *invisible latches*. The latches remaining in the abstract model are called *visible latches*. The initial abstract model is created by making the latches present in the property as visible, and the rest as invisible.

The abstract model is then passed to a model checker, typically BDD-based, such as SMV. Localization reduction is a *conservative* over-approximation of the original circuit for reachability properties. This implies that if the abstraction satisfies the property, the property also holds on the original circuit. The drawback of the conservative abstraction is that when model checking of the abstraction fails, it may produce a counterexample that does not correspond to any concrete counterexample. This is called a *spurious counterexample*.

In order to determine if the counterexample can be simulated on the concrete model, a Bounded Model Checking instance is typically formed: the concrete transition relation for the design and the given property are jointly unwound to obtain a Boolean formula. The number of unwinding steps is given by the length of the abstract counterexample. The Boolean formula is then checked for satisfiability using a SAT procedure [28]. The transitions in the abstract trace are sometimes added to reduce the search space. The disadvantage is that other counterexamples of the same length may only be detected with additional refinement. If the instance is satisfiable, the counterexample is real and the algorithm terminates. If the instance is unsatisfiable, the abstract counterexample is spurious, and *abstraction refinement* has to be performed.

The basic idea of the abstraction refinement technique is to create a new abstract model which contains more detail (e.g., more visible latches) in order to prevent the

spurious counterexample. This process is iterated until the property is either proved or disproved. There are numerous methods to refine the abstraction. If the abstract counterexample is used for refinement, the process is known as the *Counterexample Guided Abstraction Refinement* framework, or CEGAR for short [19,2,9,15,28].

Thus, successful application of abstraction refinement with localization reduction usually requires three components:

(i) A BDD-based model checker that has enough capacity for the abstract model,

(ii) a Bounded Model Checker with enough capacity to perform the simulation of the abstract trace,

(iii) a way to refine the abstraction in case the simulation fails.

In practice, despite of the abstraction, the first step often turns out to be the bottleneck, especially if the property depends on many latches.

This paper proposes the use of a technique commonly applied by many hardware engineers in an informal and manual setting: If a design is too complex for either simulation or verification, engineers cut or partition the circuit. Formally, this corresponds to removing parts of the circuit and replacing the missing signals by non-deterministically chosen inputs. This cut-point need not necessarily remove latches, and also may preserve logic dependent only on latches that were removed. The resulting circuit is an over-approximation of the original circuit with respect to safety properties.

Contribution

This paper proposes to use cut-point insertion [18] in order to compute an abstract model $\hat{M}$ with two features: 1) $\hat{M}$ over-approximates M, and thus, safety properties are preserved, and 2) we can syntactically (and thus, efficiently) identify a completeness threshold $\mathcal{CT}$ that is small enough to allow BMC with bound $\mathcal{CT}$. Thus, if no counterexample is found, we can conclude $M \models \phi$. If a counterexample is found, we check if it is spurious. If so, the cut-points are refined in order to eliminate the spurious trace. Similar to [22], we use the proof of unsatisfiability of the failed simulation run for refinement.

We therefore can omit the BDD-based model checker in the abstraction refinement loop, and rely on BMC as the only reasoning engine. This allows proving many properties with BMC only.

Related Work

Baumgartner et al. [5] perform a structural analysis similar to the one used for this paper in order to obtain a completeness threshold. In contrast to the algorithm proposed in this paper, an abstraction of the circuit in order to obtain a smaller completeness threshold is not applied. The results are extended in [4].

The concept of the completeness threshold for BMC was introduced in [17]. A completeness threshold for arbitrary LTL properties is given in [11]. Optimizations

to the diameter test that take the predicates in the property into account are given in [1].

Another popular technique to obtain a complete version of BMC is to use BMC to prove an inductive invariant [26]. The technique uses constraints to enforce simple (i.e., loop free) paths that are similar to the constraints used to perform recurrence diameter tests.

Somenzi et al. [20] use such constraints to obtain a complete BMC to be used on an abstract model in an abstraction refinement framework. As noted in [20], the depth that has to be searched using BMC can be exponentially larger than the reachability diameter.

Numerous methods have been proposed to refine an abstraction done by localization reduction. In [13], Clarke et al. propose the use of ILP solvers and machine learning techniques to choose a suitable set of latches for the abstract model. Details on how to improve the simulation step beyond the basic BMC instance are given in [3].

In [8], Chauhan et al. propose to analyze the conflict graph of the failed BMC run to obtain refinement information. A similar approach is used by McMillan [22]: the unsatisfiable core of the failed BMC run is analyzed to obtain the new set of latches used for localization reduction. The abstract model is verified using BDDs.

The first complete model checking approach based on SAT without any abstraction is presented by McMillan in [21]. A SAT solver is modified to perform pre-image computation. The approach enumerates states in the pre-image. Explicit state enumeration is avoided with an enlargement of the assignment which is derived from the conflict graph.

In [23], McMillan presents the use of interpolants in order to obtain a complete model checker based on a BMC-like reasoning engine.

Outline

In section 2, we provide background information about bounded model checking, the completeness threshold, localization reduction, and automatic abstraction refinement. We describe the abstraction we apply in section 3. Experimental results are reported in section 4.

2 Background

2.1 The Completeness Threshold and the Diameter

Let M denote a finite transition system defined by a finite set of states S, a set of initial states $I \subseteq S$, and a transition relation $R \subseteq S \times S$. By $M \models \phi$ we denote that any computation of M satisfies the property ϕ, and by $M \models_k \phi$ we denote that all computations of length k or less do not violate ϕ.

Definition 2.1 The *Completeness Threshold* [17], denoted by $\mathcal{CT}$, for a finite transition system M and a property ϕ, is any natural number such that if there is no

computation of length $\mathcal{CT}$ that violates ϕ, ϕ holds for any computation done by M:

$$M \models_{\mathcal{CT}} \phi \longrightarrow M \models \phi$$

If $M \models \phi$, then the smallest such $\mathcal{CT}$ is 0, and otherwise it is the length of the shortest counterexample. Thus, computing the smallest $\mathcal{CT}$ is as hard as determining if $M \models \phi$ holds. In practice, one therefore aims at computing over-approximations of the smallest $\mathcal{CT}$.

Definition 2.2 The *Diameter* of a finite transition system M, denoted by $d(M)$, is the length of the longest shortest path (defined by its number of edges) between any two reachable states of M.

Definition 2.3 The *Initialized Diameter* of a finite transition system M, denoted by $d^I(M)$, is the length of the longest shortest path from any initial state to any reachable state of M.

It was already observed in [6] that $d(M)$ is a sufficiently large bound to prove properties of the form **AG**p. This bound can be improved by using the initialized diameter $d^I(M)$. A bound for properties of the form **AF**p was identified in [17]. A method to compute a $\mathcal{CT}$ for arbitrary LTL properties is found in [11].

Computing the Diameter

Testing if a particular k is the diameter corresponds to a QBF instance. Despite of the progress QBF solvers made, attempts to solve such instances have failed so far. Biere et al. suggested in [6] the use of SAT to compute the recurrence diameter, which is an over-approximation of the diameter. However, for most interesting circuits, the recurrence diameter is either too large or too hard to compute. Mneimneh and Sakallah [24] modify a SAT solver to compute the diameter by path enumeration.

2.2 Over-Approximating the Diameter with Structural Analysis

Model checking is frequently applied to circuits, which are typically given as a net-list. Baumgartner et al. [5] suggest to exploit the structure of these net-lists in order to compute an over-approximation of the diameter.

Definition 2.4 A *Net-list* is a directed graph (V, E, T), where V is a finite set of vertices, $E \subseteq V \times V$ is the set of edges, and $T(v)$ is the type of the vertex $v \in V$. The type is one of and (AND-gate), inv (inverter), reg (register), inp (primary input). The in-degree of vertices of type and is at least one, of type inv and reg exactly one, and of type inp exactly zero.

Notation

Given two vertices v_1 and v_2, we write $v_1 \xrightarrow{E} v_2$ iff $(v_1, v_2) \in E$, and $v_1 \overset{E}{\rightsquigarrow} v_2$ iff there is a path from v_1 to v_2 in E.

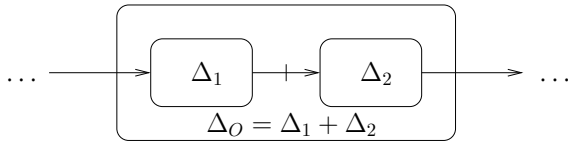

Fig. 1. Sequential composition of two components. A bound for the diameter of the composition is the sum of the individual diameters.

We write $v_1 \overset{E}{\rightsquigarrow}_G v_2$ if $T(v_1) = T(v_2) = \mathsf{reg}$ and there is a path from v_1 to v_2 in E that only goes through vertices (gates) of type $\{\mathsf{and}, \mathsf{inv}\}$. We require any such path to be acyclic, i.e., the logic between the latches must be combinational.

The definition of semantics for such a net-list is straight-forward. The conversion of circuits given in Verilog to such a net-list corresponds to synthesis.

Definition 2.5 The *Latch Dependency Graph* (LDG) of a net-list $N = (V', E', T)$ is a directed graph (V, E), where $V = \{v \in V' \mid T(v) = \mathsf{reg}\}$ is the set of latches in N, and there is an edge between two latches v_1 and v_2 in the LDG iff there is a path from v_1 to v_2 in N that only uses gates, i.e., $v_1 \overset{E}{\rightarrow} v_2 \iff v_1 \overset{E'}{\rightsquigarrow}_G v_2$.

Definition 2.6 A *Component* inside a circuit is a connected subgraph of the LDG. The *Component Graph* is the graph generated by replacing each component by a single vertex.

We denote the bound we derive for the diameter of a component C by $\Delta(C)$. Obviously 2^k is such a bound if k is the number of latches in C.

In [5], bounds for the diameter for various types of components are derived that are based on the structure of the component, e.g., for ROMs, constant latches, and acyclic components. In particular, it is observed that the sum of the bounds of the diameters of two components that are composed sequentially is a bound for the composition:

Theorem 2.7 *Let C_1 and C_2 be two components, and $\Delta(C_1)$ and $\Delta(C_2)$ be bounds for the diameter of C_1 and C_2, respectively. The sum of the two bounds is a bound for the diameter of the sequential composition $C_1 \rightarrow C_2$ (Figure 1):*

$$\Delta(C_1 \rightarrow C_2) = \Delta(C_1) + \Delta(C_2)$$

2.3 Abstraction via Cut-Point Insertion

Cut-Point Insertion corresponds to replacing a signal in the net-list by a new primary input [18]. The resulting circuit $\hat{M}$ is an over-approximation of the original circuit M, and a conservative abstraction for reachability properties. As already noted in [4], the completeness threshold of $\hat{M}$ is *not* a completeness threshold for M; the abstract circuit typically has a much smaller diameter. The diameter never increases by cut-point insertion.

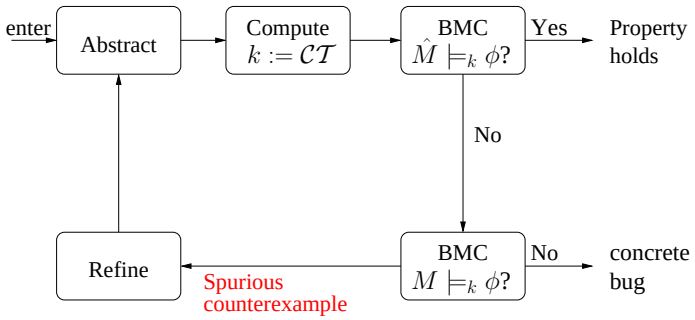

Fig. 2. Abstraction-refinement loop using BMC and the Completeness Threshold

3 A Complete BMC with Over-Approximation

3.1 Overview

Figure 2 shows an overview of the technique used in this paper. The algorithm follows the proof-based abstraction refinement loop used in [22]. We use cut-point insertion as described in section 2.3 as the abstraction technique. As initial abstraction, we insert cut-point such that all cycles in the net-list of the abstract model are eliminated. This results in a very small completeness threshold.

In contrast to most related papers that implement abstraction refinement, we do not use a BDD-based model checker to verify the abstract model $\hat{M}$. Instead, we compute a completeness threshold $\mathcal{CT}$ of $\hat{M}$. This is described in detail in section 3.2. We then perform BMC on $\hat{M}$ with bound $\mathcal{CT}$.

If the property holds on $\hat{M}$, we can conclude it also holds on M, and the algorithm terminates. Otherwise, we obtain an abstract counterexample from the BMC run. The loop then proceeds as in the related work. The refinement step is slightly different and described in section 3.3.

3.2 Computing $\mathcal{CT}$ in the Presence of Cycles

We extend the results introduced in [5] in order to obtain a completeness threshold for a larger class of designs. The main issue for the diameter over-approximation are cycles in the latch dependency graph. For cycle-free components, the most important results are summarized in section 2.2.

Thus, consider a circuit with cycles in the latch dependency graph. Such cycles are very common and typically arise from counters, or from forwarding logic in pipelined circuits. In order to over-approximate the diameter of such circuits, we define the concept of the *weighted* component graph.

Definition 3.1 The *weighted component graph* is a component graph (as in definition 2.6) in which a weight ω is assigned to each edge. We write $C_1 \rightarrow_\omega C_2$ iff there is an edge from C_1 to C_2 with weight ω. Let V_1 denote the set of latches in C_1 such that there is a path to a latch in C_2 in the LDG. Let V_2 denote these latches in C_2. The weight corresponds to the number of signals that connect V_1 and V_2.

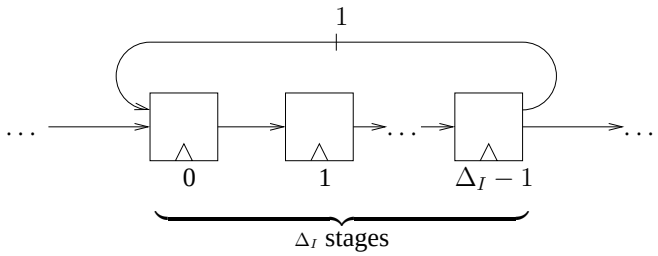

Fig. 3. The diameter of a component with a **one bit** self-loop is bounded by $2 \cdot \Delta_I$, where Δ_I denotes a bound on the diameter of the component without the loop.

As a special case, consider a circuit I with a diameter Δ_I. Without loss of generality, the circuit is represented by a pipeline with $n := \Delta_I$ stages. Now add a single-bit feedback loop (Figure 3), which forms circuit O. The signal that forms the feedback loop is computed in the last stage of I and used as input for the first stage of I. There are arbitrary connections from stage i to stage $i + 1$, but no other connections.

Claim 3.2 *The diameter of a simple pipeline pipeline with n stages and a single-bit feedback loop is bounded by $2 \cdot n$.*

We provide a proof of claim 3.2 in the appendix. This result can be generalized by eliminating the outer cycles bit by bit.

Claim 3.3 *Let $C_1, \ldots, C_n$ denote a list of pair-wise different components that a) form a cycle in the dependency graph and b) contain no sub-cycle, i.e., $C_i \rightarrow C_j \iff j = i + 1 \vee (i, j) = (n, 1)$. The diameter of the component formed by this cycle is bounded by 2^k times the sum of the bounds of the diameters of the components, where k is the weight of any edge j on the cycle:*

$$\Delta(C_1 \rightarrow_{\omega_1} C_2 \rightarrow_{\omega_2} \ldots \rightarrow_{\omega_{n-1}} C_n \rightarrow_{\omega_n} C_1) = 2^{\omega_j} \cdot \Sigma_{i=1}^{n} \Delta(C_i)$$

The proof is done by re-arranging the components such that the desired edge represents the back-cycles and then by applying Claim 3.2 k times.

Example 3.4 If the cycle consists of one component only (Figure 5), the diameter of the component and the loop is bounded by $\Delta_I \cdot 2^k$, where Δ_I denotes a bound on the diameter of the component without the loop, and k denotes the weight of the loop. A k-bit counter is an example of this case.

If the cycle has more than one component, it is desirable to break the cycle on an edge with minimal weight, as the bound is exponential in the weight of the edge. This is depicted in Figure 5: The cycle can be removed using either edge.

Figure 6 shows two cycles that share a component. Such a cycle cannot be removed with the method above. The diameter is approximated using the number of latches in all components.

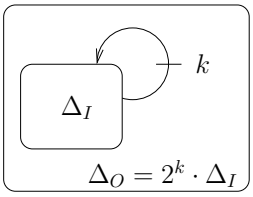

Fig. 4. The diameter of a component with self-loop is bounded by $\Delta_I \cdot 2^k$, where Δ_I denotes a bound on the diameter of the component without the loop, and k the weight of the loop.

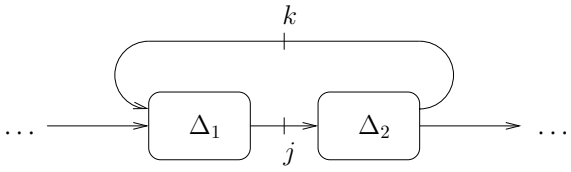

Fig. 5. The cycle can be broken using either the edge with weight k or j. The edge with minimal weight should be chosen.

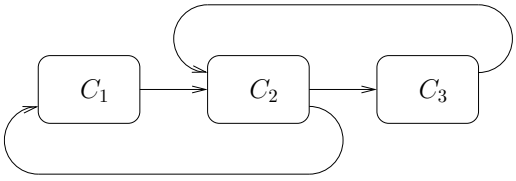

Fig. 6. Two cycles sharing a vertex C_2. The cycles cannot be removed.

3.3 Refining the Abstraction

If a spurious counterexample is detected, we obtain the unsatisfiable core of the BMC instance used for the simulation. Similar to in [22], we identify the signals that are in this core (in [22], the latches are identified). We refine the cut-points by removing those cut-points that correspond to a signal found in the core. We do not introduce new cut-points.

4 Experimental Results

We have implemented the algorithm described in section 3. We make our implementation available for experimentation by other researchers.

We apply the algorithm to various circuits already used in [16] to determine its effectiveness. The benchmarks are taken from an implementation of an out-of-order RISC microprocessor with Tomasulo scheduler. We compare the performance of the new algorithm with the performance of plain Bounded Model Checking. All experiments are performed on an Intel Xenon machine with 2.5 GHz running Linux.

Bounded Model Checking is used for refutation only, i.e., it cannot conclude that there is no error trace. Instead, it checks the property up to a given number of cycles. In [16], the property checked was consistency with a C program. We check

Benchmark	latches	bug length	Run time BMC					Run time abstraction
			min.	10	20	30	40	
ALU_PIPE1	419	2	0.2s	3.5s	26.7s	132.5s	*	0.2s
ALU_PIPE2	419	-	-	107.7s	495.1s	*	*	1.1s
RF1	1024	-	-	7.4s	30.4s	56.3s	83.4s	7.0s
RF2	1024	1	0.4s	4.6s	7.8s	23.4s	*	7.5s
ROB1	2963	-	-	2.0s	5.4s	7.8s	22.1s	225.6s
ROB2	2963	-	-	182.8s	*	*	*	310.2s
ROB3	2963	16	1.8s	1.7s	4.2s	6.3s	8.7s	6.3s
ROB4	2963	64	*	4.3s	38.3s	124.0s	387.0s	33.3s

Table 1
Experimental Results. If no bug length is given, the property holds. The run time for BMC is given for various depths. The ”min” column contains the run time for BMC for the shortest counterexample, if applicable. A star (*) denotes that the timeout of 1000s was exceeded.

safety properties instead, which is easier. Table 1 summarizes the experimental results. A short description of each circuit can be found in [16].

In conclusion, traditional BMC typically outperforms the new algorithm if the property is to be refuted. This is to be expected, as refutation is done using a regular BMC instance in the refinement loop. However, the experiments also show the benefit of the technique if the property is to be shown. In many cases, the refinement loop can show the property with a small bound.

5 Conclusion

We present an abstraction refinement loop that solely relies on BMC as its only reasoning engine. We use cut-point insertion in order to obtain an abstract model with a small completeness threshold. The completeness threshold is over-approximated with a structural analysis that permits cyclic circuits. If the abstraction is too coarse, cut-points are removed, which results in fewer spurious behavior but also a larger completeness threshold.

Our preliminary experimental results show that the technique performs well on circuits that implement a pipeline. We make our implementation available for experimentation by other researchers. [2]

Future Work

As future work, we plan to extend the algorithm that computes $\mathcal{CT}$ in order to allow arbitrary latch dependency graphs. We also plan to improve the refinement

[2] http://www.inf.ethz.ch/personal/daniekro/ebmc/

algorithm such that cut-points are also added, not only removed.

References

[1] Awedh, M. and F. Somenzi, *Proving more properties with bounded model checking*, in: R. Alur and D. A. Peled, editors, *16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science **3114** (2004), pp. 96–108.

[2] Ball, T. and S. Rajamani, *Boolean programs: A model and process for software analysis*, Technical Report 2000-14, Microsoft Research (2000).

[3] Barner, S., D. Geist and A. Gringauze, *Symbolic localization reduction with reconstruction layering and backtracking.*, in: E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science **2404** (2002), pp. 65–77.

[4] Baumgartner, J. and A. Kuehlmann, *Enhanced diameter bounding via structural transformation*, in: *Design, Automation and Test in Europe Conference and Exposition (DATE)* (2004), pp. 36–41.

[5] Baumgartner, J., A. Kuehlmann and J. A. Abraham, *Property checking via structural analysis*, in: E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science **2404** (2002), pp. 151–165.

[6] Biere, A., A. Cimatti, E. Clarke and Y. Yhu, *Symbolic model checking without BDDs*, in: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science **1579** (1999), pp. 193–207.

[7] Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation **98** (1992), pp. 142–170.

[8] Chauhan, P., E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith and D. Wang, *Automated abstraction refinement for model checking large state spaces using sat based conflict analysis.*, in: M. Aagaard and J. W. O'Leary, editors, *4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Lecture Notes in Computer Science **2517** (2002), pp. 33–51.

[9] Clarke, E., O. Grumberg, S. Jha, Y. Lu and V. H., *Counterexample-guided abstraction refinement*, in: *CAV*, 2000, pp. 154–169.

[10] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[11] Clarke, E., D. Kroening, O. Strichman and J. Ouaknine, *Completeness and complexity of bounded model checking*, in: *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science **2937** (2004), pp. 85–96.

[12] Clarke, E. M. and E. A. Emerson, *Synthesis of synchronization skeletons for branching time temporal logic*, in: *Logic of Programs: Workshop*, LNCS **131**, Springer-Verlag, 1981 .

[13] Clarke, E. M., A. Gupta, J. H. Kukula and O. Strichman, *SAT based abstraction-refinement using ILP and machine learning techniques.*, in: E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science **2404** (2002), pp. 265–279.

[14] Copty, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Y. Vardi, *Benefits of bounded model checking at an industrial setting*, in: G. Berry, H. Comon and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, number 2102 in Lecture Notes in Computer Science (2001), pp. 436–453.

[15] Das, S. and D. Dill, *Successive approximation of abstract transition relations*, in: *16th Annual IEEE Symposium on Logic in Computer Science (LICS)* (2001).

[16] Kroening, D. and E. Clarke, *Checking consistency of C and Verilog using predicate abstraction and induction*, in: *Proceedings of ICCAD* (2004), pp. 66–72.

[17] Kroening, D. and O. Strichman, *Efficient computation of recurrence diameters*, in: L. Zuck, P. Attie, A. Cortesi and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science **2575** (2003), pp. 298–309.

[18] Kuehlmann, A. and F. Krohm, *Equivalence checking using cuts and heaps*, in: *Proceedings of the 34th annual conference on Design automation (DAC)* (1997), pp. 263–268.

[19] Kurshan, R., "Computer-aided verification of coordinating processes: the automata-theoretic approach," Princeton University Press, 1994.

[20] Li, B., C. Wang and F. Somenzi, *Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure*, International Journal on Software Tools for Technology Transfer (STTT) **7** (2005), pp. 143–155.

[21] McMillan, K., *Applying SAT methods in unbounded symbolic model checking*, in: *14th Conference on Computer Aided Verification*, 2002, pp. 250–264.

[22] McMillan, K. and N. Amla, *Automatic abstraction without counterexamples*, in: *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science (2003).

[23] McMillan, K. L., *Interpolation and SAT-based model checking*, in: W. A. H. Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science **2725** (2003), pp. 1–13.

[24] Mneimneh, M. and K. Sakallah, *SAT-based sequential depth computation*, in: *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC)*, 2003, pp. 87–92.

[25] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *DAC*, 2001, pp. 530–535.

[26] Sheeran, M., S. Singh and G. Stålmarck, *Checking safety properties using induction and a SAT-solver*, in: *Third International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2000), pp. 108–125.

[27] Shtrichman, O., *Tuning SAT checkers for bounded model checking*, in: E. Emerson and A. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, Lecture Notes in Computer Science (2000), pp. 480–494.

[28] Wang, D., P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma and R. Damiano, *Formal property verification by abstraction refinement with formal, simulation and hybrid engines*, in: *DAC*, 2001, pp. 35–40.

A Proofs

Let $\mathcal{D}_i$ denote the range of values that the registers in stage i can take. Let $P_i(t) \in \mathcal{D}_i$ denote the value in pipeline stage i at time t. For $i > 0$, P_i only depends on P_{i-1}. Let f_i denote the function that represents this dependence:

(A.1) $P_i(t) = f_i(P_{i-1}(t-1))$

Let $\Gamma(t) \in \mathbb{B}$ denote the value of the feedback bit at time t. The feedback bit is computed using P_{n-1} only. We use γ_{n-1} to denote the function that represents this dependency.

(A.2) $\Gamma(t) = \gamma_{n-1}(P_{n-1}(t))$

This definition can be extended inductively to the other stages:

(A.3) $\gamma_i(x) := \gamma_{i+1}(f_{i+1}(x))$

Thus, the data value $x \in \mathcal{D}_i$ in stage i will produce $\gamma_i(x)$ as feedback bit once it arrives in the last stage.

Definition A.1 We call $\gamma_i(x)$ the *color* of stage i.

Lemma A.2 *For all $0 \leq i \leq j < n$, the color of stage j at time t is the color of stage i at time $t - j + i$:*

$$\gamma_{n-1}(P_{n-1}(t)) = \gamma_i(P_i(t - j + i))$$

The proof of this lemma is easily done by induction on $n - 1 - i$.

Let $\iota(t)$ denote the value of the primary inputs in cycle t. The value computed for the first stage depends only on the feedback bit and these primary inputs. Let the value computed for the first stage be denoted by $f_0(\iota, \gamma) \in \mathcal{D}_0$.

Lemma A.3 *The value in any stage at time $t \geq n$ only depends on primary inputs and on the color of the same stage n cycles earlier.*

Proof. First, observe that $P_i(t)$ with $t \geq n$ only depends on the value of primary inputs during cycle $t - i - 1$ and on the value of the feedback bit at time $t - i - 1$:

(A.4) $P_i(t) = f_i \circ \ldots \circ f_0(\iota(t-i-1), \Gamma(t-i-1))$

By expanding the definition in Eq. A.2, we obtain:

(A.5) $P_i(t) = f_i \circ \ldots \circ f_0(\iota(t-i-1), \gamma_{n-1}(P_{n-1}(t-i-1)))$

We can use Lemma A.2 with $j = n-1$ to rewrite Eq. A.5 and obtain:

(A.6) $P_i(t) = f_i \circ \ldots \circ f_0(\iota(t-i-1), \gamma_i(P_i(t-n)))$

The color of stage i at time $t-n$ is $\gamma_i(P_i(t-n))$, which concludes the claim. □

We now show the main claim 3.2.

Proof. [Claim 3.2] We show that we can bring the pipeline into any reachable state s within $2 \cdot n$ clock cycles or less.

If s is reachable, there must be a path $s_0, \ldots, s_t$ from an initial state s_0 to state $s = s_t$. Let t denote the length of the path. If $t \leq 2 \cdot n$, there is nothing to show.

Otherwise, we bring the circuit into state s as follows: (1) We start with the same initial state s_0. (2) In the next n cycles, by picking appropriate primary inputs, we bring the pipeline into a state such that the colors at time n match the colors in state s_{t-n}. Such primary inputs exist, or otherwise, s_{t-n} is not reachable. (3) In cycles n to $2n-1$, we bring the pipeline into the desired state by simply re-playing the primary inputs used to obtain $s_{t-n+1}, \ldots, s_t$. This is sufficient according to lemma A.3. □

Authors

A

Awedh 51

F

Franzén 23

G

Geist 65
Ginzburg 65

J

Jehle 9
Johannsen 9

K

Kroening, D. ... 75

L

Lange 9
Lustig 65

R

Rabinovitz 65
Rachinsky 9

S

Shacham 37,65
Somenzi 51

T

Tzoref 65

Y

Yorav 37