# BMC'04

# Second International Workshop on Bounded Model Checking

## Preliminary Proceedings

18. July 2004

Boston, MA, USA

# Preface

These are the *preliminary proceedings* of the second international workshop on Bounded Model Checking (BMC'04), which is affiliated to the 16th International Conference on Computer-Aided Verification (CAV'04) in Boston, MA, USA. The workshop takes place immediately after the conference on Sunday, July 18, 2004, and besides the presentations of the papers collected in this volume also features an invited talk by Bart Selman. Ouf of 9 submissions the program comittee selected 6 papers, which after the workshop will be published electronically as a special volume of Electronic Notes on Theoretical Computer Science (ENTCS).

## Organizers


Armin Biere
*ETH Zürich, Switzerland*

Ofer Strichman
*Technion, Israel*


## Program Comittee


Per Bjesse
*Synopsys, USA*

Alan Hu
*UBC, Canada*

Alessandro Cimatti
*IRST, Italy*

Sharad Malik
*Princeton, USA*

Koen Claessen
*Chalmers, Sweden*

João Marques Silva
*Lisbon, Portugal*

Ranan Fraer
*Intel, Israel*

Ken McMillan
*Cadence, USA*

Danny Geist
*IBM Israel*

Fabio Somenzi
*Boulder, USA*

Yunshan Zhu
*Synopsys, USA*


Armin Biere, Ofer Strichman

Zürich, Haifa, June 2004

# Contents

# SAT-based Induction for Temporal Safety Properties

Roy Armoni [a,1] Limor Fix [a,1] Ranan Fraer [a,1]
Scott Huddleston [b,1] Nir Piterman [a,1] Moshe Y. Vardi [c,2,3]

[a] *Design Technology – Intel, Haifa, Israel*

[b] *Desktop Product Group – Intel, Hilsboro, Oregon*

[c] *Dept. of Computer Science, Rice University*

**Abstract**

The work presented in this paper addresses the challenge of fully verifying complex temporal properties on large RTL designs. Windowed induction has been proposed by Sheeran, Singh, and Stalmarck as a technique augmenting Bounded Model Checking for unbounded verification of safety properties. While induction proved to be quite effective for combinational properties, the case of temporal properties was not handled by previously known methods. We introduce *explicit induction*, a new induction scheme targeted to temporal properties, and to interactive development of inductive proofs. The innovative idea in explicit induction is to make the induction scheme an explicit part of the specification, where it can be easily controlled, using a highly expressive language like ForSpec. We show how explicit induction was implemented with minor modifications in the ForSpec compiler and in Thunder, a bounded model checker. Finally, we describe how explicit induction was used for verifying large control circuits with extensive feedback in the Pentium$^{\text{TM}}$4 processor. The circuits verified by explicit induction are orders of magnitude larger than those verifiable by traditional model checking approaches.

*Key words:* SAT, induction, windowed induction, safety

## 1 Introduction

The general aim of formal verification is to provide compelling evidence of the correctness of a system in the form of a mathematically precise argument

[1] Email: firstname.lastname@intel.com

[2] Email: vardi@cs.rice.edu

[3] Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, and ANI-0216467 by BSF grant 9800096, and by a grant from the Intel Corporation.

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

showing that the system (implementation) satisfies a collection of required properties (specification). In model checking, we verify the correctness of finite-state systems with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior [6]. Model-checking has two major advantages, namely, it is fully automatic, and in the case of failure produces a counterexample (an erroneous execution of the system).

The introduction of symbolic model checking based on BDDs [4,10] has increased the capacity of model checking and made it a standard in hardware industry [1]. BDDs are a canonic representation of Boolean functions and are used to represent sets of states and transitions of the model. BDD-based model checkers compute the set of reachable states (or sometimes analyze cycles) to ensure that there are no disallowed behaviors. In spite of the increased capacity, it soon became apparent that *state explosion* is still a problem. A major breakthrough has been the introduction of bounded model checkers [2]. Bounded model checking is based on the representation of computation paths falsifying the specification in the form of a Boolean satisfiability problem. The usage of bounded model checking increased the size of models handled by model checkers; however, at a price. We no longer get a fully certified answer to the verification problem but rather assurance that there are no counterexamples of a given length. This observation makes bounded model checking especially adequate for bug hunting. Still there is a limit on the size of bounds handled by bounded model checkers, leaving us with lack of complete assurance in the correctness of the design under verification.

In practice, most specifications are *safety* properties. A property is called *safety* if we can deduce that it is false by examining a finite computation path. Combinational properties of the form `ALWAYS` $p$, also called *invariants*, are a particular case of safety properties that have two distinctive characteristics:

- Every safety property can be reduced to an invariant by a compilation process described below [8].
- Some invariants can be fully proved with a powerful technique, called *induction* [13].

Besides being a complete proof technique, induction has a better capacity than bounded model checking, as it has to unroll the model only to a small depth. The following paragraph explains how the induction works.

### *1.1 Induction for Invariants*

Traditional mathematical induction can be used to prove that a property $P(n)$ holds for all nonnegative integers $n$. An induction proof consists of proving the following two subgoals:

- Prove that $P(0)$ is true.
- Prove that forall $k$, $P(k)$ implies $P(k+1)$.

In formal verification, induction has been used to prove an invariant $P$ in a transition system by showing that $P$ holds in the initial states of the system and that $P$ is maintained by the transition relation of the system [9]. In many cases $P$ is not inductive by itself, and one has to find a strengthening of $P$ that is inductive.

More formally, let $M = (S, S_0, T)$ be a transition system, where $S$ is a set of states, $S_0 \subseteq S$ is a set of initial states and $T \subseteq S \times S$ is a transition relation. For simplicity of presentation, we relate to sets of states as predicates, e.g., by using the characteristic function of the set. The classical induction methodology for proving $P$ is based on manually finding a property $Q$ (the *induction hypothesis*) such that $Q \Rightarrow P$ and proving the following two subgoals:

- The initial states of $M$ satisfy $Q$: forall states $x_0$, we have $S_0(x_0) \Rightarrow Q(x_0)$
- $Q$ is maintained by the transition relation: forall states $x_0$ and $x_1$ we have $Q(x_0) \wedge T(x_0, x_1) \Rightarrow Q(x_1)$

This classical method is known to be theoretically sound and complete, where theoretical completeness is demonstrated by having the property $Q$ describe the set of reachable states of $M$. Note that induction hypotheses are typically much simpler than a full reachable state description. When it succeeds, induction is able to handle larger models than bounded model checking, since the induction step has to consider only paths of length 1, whereas bounded model checking needs to check sufficiently long paths to get a reasonable confidence [3].

### *1.2 Windowed Induction*

In many cases, constructing an inductive invariant for simple induction is not feasible. Windowed induction is a modified induction technique, which can considerably simplify finding inductive invariants for proofs on hardware models. Mathematically, windowed induction with window size $N \geq 0$ consists of the following two steps:

- Prove that for $0 \leq k \leq N$, $P(k)$ is true.
- Prove that forall $k$, $(P(k) \wedge \ldots \wedge P(k+N)) \Rightarrow P(k+N+1)$.

Windowed induction proofs in a hardware system are realized as follows [13]. To prove that $P$ is an invariant of system $M$, we do the following:

(i) Manually find a strengthening $Q$ of $P$ for which $Q$ implies $P$. Typically we choose $Q$ to be $P \wedge \langle \text{something} \rangle$.

(ii) Find an $N$ for which the following two proofs are achievable:
   (a) Base: $Q$ holds in all paths of length $N$ starting from an initial state:

$$S_0(x_0) \wedge T(x_0, x_1) \wedge \ldots T(x_{N-1}, x_N) \Rightarrow Q(x_0) \wedge Q(x_1) \wedge \ldots \wedge Q(x_n)$$

   (b) Step: For an arbitrary path of length $N+1$, if $Q$ holds in the first

$N+1$ states, then it holds in state $N+2$ too

$$T(x_0, x_1) \wedge \ldots \wedge T(x_N, x_{N+1}) \wedge Q(x_0) \wedge \ldots \wedge Q(x_N) \Rightarrow Q(x_{N+1})$$

This method is also known to be sound and complete. Even without strengthening $P$, if we restrict the induction step only to loopfree paths, completeness can be proved by choosing $N$ to be the recurrence diameter of the transition system $M$, i.e., the maximum length of a loopfree path in $M$. The advantage of windowed induction over classical induction is that it provides the user with two ways of strengthening the induction hypothesis: strengthening the invariant $Q$ or lengthening the window $N$. (For simplicity, we do no mention the loopfreeness condition in the rest of our discussion, but it is implemented in our tool.)

Windowed induction is used in [13], and is considered more abstractly in [5]. The formal verification environment in Intel offers this induction scheme as an automatic mode in the SAT-model checker Thunder. The bound $N$ is iteratively increased until either the proof succeeds or a given limit is reached. Windowed induction and standard induction have the same theoretical capabilities, but windowed induction often permits much simpler induction hypotheses. In many cases the size of windows is relatively small. As with simple induction, we get the best of both worlds: we get a correctness proof and we get the ability to handle very large models.

Intuitively, pipelines in sequential hardware circuits are why windowed induction proofs can use simpler induction hypotheses than non-windowed induction proofs of the same property. When a property $P$ of interest depends on a pipeline of depth $d$, a windowed induction with window size $d$ can sometimes prove $P$ inductively without strengthening $P$. But a comparable standard induction proof in this case generally has to strengthen $P$ to express many internal invariants on the pipeline in order for the proof to succeed.

The techniques in [13] have been used successfully at Intel, as they proved to be quite effective for verifying combinational properties. One benefit of this approach is that it automates much of the induction mechanism, including automatically searching for a working induction window size.

### *1.3 Implicit Induction for temporal properties*

When a property $P$ is not combinational, but rather a complex temporal specification such as a typical formula that is written using the specification language ForSpec, it may not be obvious how to prove $P$ by induction. As mentioned, these assertions are often safety properties. The ForSpec compiler synthesizes $P$ into an automaton $A_P$ and an invariant $Z_P$ [14,8] such that for every transition system $M$:

$$M \models P \text{ iff } M_P \models \texttt{ALWAYS}\ Z_P$$

Where $M_P = M \parallel A_P$ denotes the synchronous composition of $M$ and $A_P$. The above observation about the behavior of the ForSpec compiler, immediately suggests that for a safety property $P$, the classical induction methodology may be used to prove $M \models P$ by proving that $M_P \models$ `ALWAYS` $Z_P$.

As an example, consider the ForSpec property `ALWAYS` $\neg(f, f)$, which forbids two consecutive occurrences of $f$, where $f$ is a combinational property. When compiling its negation, the property `EVENTUALLY` $(f, f)$, we can get, for example, the three-state automaton in Figure 1. This automaton can cycle in its initial state $s_0$ or move nondeterministically to $s_1$ upon receiving the input $f$. A second occurrence of $f$ is needed to move from $s_1$ to the accepting state $s_2$, while an occurrence of $\neg f$ brings us back to $s_0$. The output $Z$ marks the accepting state of this automaton, such that $Z = 0$ if and only if the original property `ALWAYS` $\neg(f, f)$ fails. If we manage to prove `ALWAYS` $Z$ by induction, then we have also a complete proof for `ALWAYS` $\neg(f, f)$.

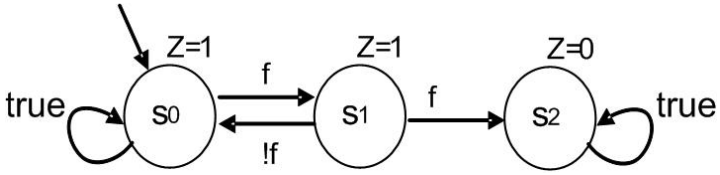


Fig. 1. Accepting automaton for `EVENTUALLY` $(f, f)$ - the negation of `ALWAYS` $\neg(f, f)$

This approach is implemented at Intel on top of Thunder, a bounded model checker [7]. When invoking the induction algorithm, the tool searches for a large enough window size $N$ for which both requirements (ii-a) and (ii-b) hold. Completeness guarantees that such an $N$ exist, even without strengthening the invariant. In practice, the induction succeeds with a reasonably small $N$ for many combinational invariants. For temporal properties expressed in ForSpec, however, the tool typically fails to complete the proof. The problem is both algorithmic and methodological, as we now explain.

On the algorithmic side, the approach suffers from a serious capacity issue. Consider again the property $P =$ `ALWAYS` $\neg(f, f)$ and the automaton in Figure 1. Assume that this property can be proven by classical induction, i.e., the initial states of $M$ satisfy $\neg(f, f)$ and $\neg(f, f)$ is maintained by the transition relation. Thus, this property passes with a window $N = 1$. However, `ALWAYS` $Z$ cannot be proven by induction with a small window $N$. For every small $N$, there is a path of length $N + 1$ failing the induction step (the path looping $N - 1$ times in $s_0$ before taking a transition to $s_1$ and then to $s_2$). Although this path contains loops in the automaton $A_P$ alone, it can well be a loopfree path in the product $M_P = M \parallel A_P$. In the worst case, the minimal window $N$ for which the induction succeeds is the recurrence diameter of the system $M$ (the minimal length of loopfree paths in $M$), which usually exceeds the capacity of the tool. (It might seem that the problem is caused by the specific automaton used in the example, but this is not the case.)

One way to reduce the window size is to strengthen the invariance $Z_P$

manually. This, however, requires expressing an invariant over the augmented design $M_P$. While the user can be expected to have an understanding of the internal details of the system $M$, the user cannot be expected to have an understanding of the internal details of the automaton $A_P$, which is the output of the ForSpec compiler. The user can even less be expected to understand the interaction of $M$ with $A_P$. Thus, requiring the user to generate invariances of $M_P$ is not realistic. This suggests that implicit induction, even with manual intervention, cannot be effectively used for verifying temporal ForSpec properties.

### *1.4 Explicit Induction for Temporal Properties*

This limitation is in fact what motivated our work. We want to perform the inductive reasoning directly on the original temporal formula, rather than on the results of its compilation. To this end, we encode the induction scheme *explicitly* as part of the specification. A failure to prove the induction produces a meaningful counterexample, which reflects the real reason of the failure and is not due to compilation artifacts anymore. Understanding the induction failure is a crucial hint for strengthening the inductive property.

In our opinion, this is the only effective way to allow the manual guidance of the user in the iterative process of finding the right induction hypotheses. In this respect, we take the approach of [13] one step further, and to our knowledge this is the first attempt to perform induction directly on temporal properties.

To be more precise, consider again a transition system $M = (S, S_0, T)$ and denote by $uninit(M)$ the non-initialized model $(S, S, T)$ in which every state in $S$ is an initial state.

Consider a ForSpec formula $P$. We say that $P$ is *bounded* if there exists $k$ such that for every path $\pi$, the truth value of $P$ on $\pi$ can be determined by considering the prefix of $\pi$ of length $k$. For example, the truth value of $a \wedge \ [5]\ b$ (that is, $a$ holds now and $b$ holds after 5 time units) can always be determined by considering a prefix of length 6. Similarly, the truth value of $(a[3], b[2])$ `TRIGGERS NEXT` $c$ (that is, 3 occurrences of $a$ followed by 2 occurrences of $b$ must be followed by an occurrence of $c$) can be determined by considering a prefix of length 6. On the other hand, there does not exist a bound $k$ such that the truth value of the formulas $a$ `UNTIL` $b$ or $(a[2]b^*c)$ `TRIGGERS NEXT` $d$ (that is, 2 occurrences of $a$ followed by some number of occurrences of $b$ and then a $c$ must be followed by $d$) can be determined by considering prefixes of length $k$. Suppose that $P$ is a bounded ForSpec specification. Proving `ALWAYS` $P$ by explicit induction requires the following:

(i) Manually find a strengthening $Q$ of $P$ for which `ALWAYS` $Q$ implies `ALWAYS` $P$. Again, usually $Q$ is chosen to be $P \wedge \langle \text{something} \rangle$.

(ii) Find an $N$ for which the following two proofs are achievable:
   (a) Base: Prove that $M \models$ `ALWAYS`$[0, N]\ Q$

(b) Step: Prove that $uninit(M) \models (\texttt{ALWAYS}[0, N]\ Q) \Rightarrow [N+1]\ Q$

For bounded formulas $Q$, both (ii-a) and (ii-b) can be proved using bounded model checking, as we discuss later. For readers not familiar with the ForSpec language, the formula $\texttt{ALWAYS}[0, N]\ Q$ means that $Q$ holds in the first $N+1$ states of a path. Similarly, $[N+1]\ Q$ means that $Q$ holds in the $N+2$-nd state of a path.

Now, one can easily see that the requirements (ii-a) and (ii-b) in explicit induction are the analogs of their counterparts in windowed induction. As a consequence the explicit induction is sound and complete too (recall the loopfreeness default constraint). The major difference between explicit induction and implicit induction is that the induction is no longer an internal algorithm inside the model checker. Rather, the induction scheme becomes an integral part of the specification, where it can be easily controlled by the user, using the expressiveness of the ForSpec specification language. This combines the qualities of windowed induction with the ability to prove properties that are more complex than simple invariants.

## 2 Tool Issues

This section examines the tool support necessary to implement the induction checks (ii-a) and (ii-b) described in Subsection 1.4. Note first that (ii-a) comes down to checking the assertion $\texttt{ALWAYS}\ Q$ until bound $N$ (see discussion below for the impact of formula depth on the bound). This is a plain bounded model checking problem.

Similarly (ii-b) can be reduced to performing bounded model checking for the assertion $\texttt{ALWAYS}\ Q$ with bound exactly $N+1$ (again, see discussion below), with the assumption $\texttt{ALWAYS}[0, N]\ Q$, on an uninitialized version of $M$. This means that the tool performs bounded model checking at bound exactly $N+1$ on a model that is formed by composing three smaller models:

- the uninitialized model $uninit(M) = (S, S, T)$, derived from the original model $M = (S, S_0, T)$,
- the initialized automaton of the assertion $\texttt{ALWAYS}\ Q$,
- the initialized automaton of the assumption $\texttt{ALWAYS}[0, N]\ Q$.

While the model $M$ has to be uninitialized for the induction to be sound, the two automata compiled by ForSpec have to be initialized as in a regular run. For instance, the counter used in the $[0, N]$ time window of the assumption needs to start from 0, otherwise our check does not implement correctly the inductive step.

To sum up, we have reduced (ii-b) to another instance of bounded model checking where the property $\texttt{ALWAYS}\ Q$ is checked exactly at bound $N+1$, and the initial constraints are built selectively only from the two ForSpec automata, but not from the model $M$ itself, or from other ForSpec properties.

To solve this last issue, we offer to the user a new ForSpec keyword, `INDUCTION_HYPOTHESIS`, that he can use to mark the assumption `ALWAYS`$[0, N]\, Q$. This way, one can distinguish between assumptions that strengthen the induction proof and regular assumptions that specify the interface with neighbor RTL blocks. Based on the new keyword, the ForSpec compiler marks every one of the automata it generates as an assertion, assumption, or induction hypothesis. This information is passed to the model checker that makes sure to use only the initial constraints from the assertion and the induction hypothesis.

Finally, note that the bounds $N$ and $N+1$ used in the two checks above are appropriate for the case where $Q$ is a combinational property. We mentioned earlier that $P$ (and hence $Q$) can be a bounded safety property. In this case, the bounds used depend on the length of the time windows employed in $P$. Usually, we end up choosing an offset $k$, such that the induction base (ii-a) is checked at bound $N + k$, while the induction step (ii-b) is checked at bound $N + k + 1$.

## 3 Usage Methodology

In this section we cover our methodology for working with explicit induction. From a usage point of view, there are three primary differences between explicit induction and implicit induction:

- The user must write the induction hypothesis explicitly, where the implicit induction builds it automatically from the assertions.
- The user must supply a window size in the explicit induction, while the implicit induction searches for a working induction window size by trying increasingly larger sizes.
- In implicit induction, the invariant is the result of automatic translation of the property. Hence, the user may find it difficult to strengthen the invariant (as explained above) and his most probable strategy would be to increase the window size. In explicit induction, the strengthening can be achieved by changing both the invariant and the window size.

The ForSpec directives needed to express explicit induction proofs are `ASSERT`, `ASSUME`, and `INDUCTION_HYPOTHESIS`. The keyword `ASSUME` is used to give auxiliary assumptions (e.g., assumptions about inputs etc.). The explicit induction technique does not depend on the usage of ForSpec, the ForSpec constructs described below help expressing and maintaining the induction hypotheses.

We exploit the "block template" construct in ForSpec to generate related formulas for a given temporal property $Q$. For instance, we define a block template mk_induction_specs$(Q, N)$ that generates for a given property $Q$, and a window $N$ the induction hypothesis and the assertion necessary for the proof:

mk_induction_specs($Q, N$) := {

upto_cycle_NN := ALWAYS$[0, N]$ $Q$;

at_all_times := ALWAYS $Q$;

}

The explicit induction directives for $Q$ would then look as follows in ForSpec:

myspec := mk_induction_specs($Q, N$); // instantiate the block

`INDUCTION_HYPOTHESIS` myspec/upto_cycle_$N$;

`ASSERT` myspec/at_all_times;

We then check these two directives in a model checking run of $N+1$ cycles. Not only does the block template give us a compact notation, it also keeps the low level formulas denoting assertions and inductive hypotheses synchronized. They refer to the same property $Q$ and the same bound $N$, which prevents false positives.

In some cases it is convenient to use a different window size for different induction hypotheses. This provides useful insight into the pipeline/logic depth that each property depends on, and also helps select minimal sufficient model checker bounds to control complexity. It is a strength of this methodology that it is flexible enough to use either independent or identical induction window sizes for different specs.

For any property that can be proved inductively, there will be some induction window size for which it and all larger window sizes yield a successful proof, and all smaller window sizes fail with a counterexample in the formal model. Initially it is not known what window size is needed, so starting with larger window sizes can be beneficial. On the other hand, shorter window sizes find counterexamples more quickly, and shorter counterexamples are easier to debug.

As a general rule, we estimate an induction window size $N$ that might work and try to prove the property $Q$. If we get a failure, we examine the counterexample. If the counterexample looks like it could be due to out-of-sync initialized pipelines, the induction window size needs to be increased. If the counterexample looks like it is caused by some state in the formal model that should not be reached in the actual circuit, the inductive hypothesis probably needs strengthening. The strengthened hypothesis typically adds a new constraint that forbids some problematic state combination that contributes to the counterexample.

Assertions that depend on simple pipelines demonstrate the advantage of windowed induction over "depipelining", i.e., of turning a windowed induction hypothesis into a larger invariant for simple induction. Where a windowed induction hypothesis can simply reference values of interest at the ends of

the pipelines, a simple induction invariant must explicitly express a constraint at every pipeline stage. When the logic driving the assertion to prove is more complex than simple pipelines, the effort and cost to construct a simple induction invariant is much higher.

This methodology also allows an intermediate approach between pure simple induction and long induction windows. For simple pipelines amenable to depipelining, one can partition the pipeline into two (or more) pieces of roughly equal length, and express invariants at just the partition points and end. For very long pipelines this can roughly halve the required induction window size with only a small depipelining cost in invariant construction. It is a strength of any windowed induction technique, including ours, that this trade off is available.

# 4 Application to the Lock Protocol in the Pentium™4 Processor

The lock protocol is used in the Pentium™4 to allow different threads to execute atomic operations on several shared resources. The lock protocol is important to verify because it interacts subtly with several other microarchitectural features, making its functional correctness crucial. The lock protocol interacts with the cache coherence protocol, as well as with several other performance optimizations.

One basic requirement for such a protocol is mutual exclusion: no resource can be locked by two threads at once. This property is expressed by a ForSpec formula of the form `ALWAYS` $a$ `TRIGGERS` $b[k]$, where $a$ and $b$ are certain signals of the design and $k$ is an integer. Note that this property is bounded. Using bounded model checking this property was proved on all traces of up to 50 cycles. Given the importance of this property, it was necessary to get a full unbounded proof and we used explicit induction for that purpose.

When trying to prove the mutual exclusion property, we quickly get induction failures. The model checker provides a witness trace for which the inductive step does not hold. Usually this is due to starting in states that are unreachable in the real model. For instance, the control part of the protocol is modeled as a finite state machine. Some induction failures are traces that include concurrent occurrence of certain events that cannot actually happen together in the real model. Adding a simple induction hypothesis eliminates such trivial failures.

Developing the inductive proof is therefore an iterative process, where we keep strengthening the induction hypothesis based on the failures of previous attempts to establish induction. Overall, we had to add about twenty constraints before the hypothesis was sufficiently strong to establish induction. The window size of the inductive proofs differed from property to property. Only three properties were provable with a window size of one cycle. All the other properties were proved with a window size of six to twelve cycles.

The model we verified is quite large, containing about 12,000 state elements. This is considerably beyond the capacity (a few hundred state elements) of BDD-based model checkers. The verification effort for the proved properties described here took three to six person-months. Most induction-step runs completed within 20 minutes, checking 36 steps, using under 600M of memory. All induction-step runs completed within 3 hours, checking 48 steps, using under 1G memory.

## 5 Similar Approaches to Induction

Two potential alternatives invite a comparison with our approach. The most direct comparison is with the implicit induction based on [13]. This approach, as currently implemented in Thunder is incapable of inductively proving most temporal properties written in ForSpec, so no direct comparison is possible. We explained earlier the reasons for the failure of implicit induction for temporal properties. While the capacity problem cannot be eliminated by using a different compilation scheme, we believe that it can be alleviated. For example, were we to compile the properties into deterministic automata, the counterexample traces for the induction step (ii-b) would be more constrained, reducing the required window size $N$. This is a topic of further research. Even if that technology becomes available, it is possible that for very large models our approach would still be preferable to implicit induction, because of the insight into induction window lengths our approach can give on a property by property basis.

Another comparison with our approach is induction using STE instead of SAT as the bounded model checker. Induction with STE [12] has been used successfully at Intel for several years, particularly for datapath logic and floating-point property proofs. Sajid and Kaviola pioneered the extension of STE induction to a moderately complex control logic property [11]. Direct comparisons between the STE and SAT induction approaches are not easy because of tool differences. Our 12000 state element model, even reduced to the critical latches, is likely more than twice the size of the 3000 state element model in [11]. Perhaps the largest distinction between SAT induction and STE induction is the apparent difficulty or impracticality of doing windowed induction (vs. simple induction) with STE. Windowed induction using STE is theoretically possible, but it has problems with rapid variable blowup and/or *antecedent conflicts*. The work of [11] includes simple induction for exactly this reason.

## 6 Results

The concepts presented here have been implemented in the formal verification tool suite at Intel. Only minor modifications were required to the ForSpec compiler and to Thunder, our SAT based model checker. The explicit in-

duction approach has been successfully used in the DPG Formal Verification Group. In particular, the most impressive application was the full verification of the lock protocol described above. The size of the model,12000 state elements, and the complexity of the protocol speak for themselves. But beyond the quantitative data, there is the impact of a new methodology that can address verification problems that cannot be handled with the existing technologies.

The use of windowed induction seems to be a critical factor in enabling successful proofs of these properties. If it was necessary to *depipeline* our windowed induction hypotheses into hypotheses sufficient for simple induction, a reasonable estimate is that spec volume would increase by at least a factor of 10, effort at least triple, and comprehensibility and maintainability would be considerably reduced.

## 7 Summary

The methodology and tool support presented in this paper address the challenge of fully verifying complex temporal properties on large RTL designs. The methodology advocated here is interactive development of inductive proofs. There is much ongoing research for automating induction proofs, but no satisfactory technique has been found, even for combinational properties. We believe that for proofs of the complexity encountered in our work, user guidance is needed for finding the correct induction invariants.

At the core of our approach is the explicit induction, a new induction scheme targeted to temporal properties, and to interactive development of inductive proofs. The case of temporal properties was not adequately addressed by previously known methods. The innovative idea in the explicit induction is to make the induction scheme an explicit part of the specification, where it can be easily controlled, using a highly expressive language like ForSpec.

The current experience shows that the explicit induction is capable of handling verification problems that were previously intractable with all the existing technologies. To further push this approach, our future work will focus on automating some of the manual tasks, for instance by having the tool suggest candidates for induction invariants.

## References

[1] Beer, I., S. Ben-David, C. Eisner and A. Landver, *RuleBase: An industry-oriented formal verification tool*, in: *Proc. 33rd Conference on Design Automation* (1996), pp. 655–660.

[2] Biere, A., A. Cimatti, E. Clarke, M. Fujita and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, in: *Proc. 36th Design Automation Conference* (1999), pp. 317–320.

[3] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs*, in: *Computer Aided Verification, Proc. 11th International Conference*, Lecture Notes in Computer Science **1633** (1999), pp. 172–183.

[4] Bryant, R., *Graph-based algorithms for boolean-function manipulation*, IEEE Trans. on Computers **C-35** (1986).

[5] Claessen, K., *Induction and state machines* (1999), unpublished.

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

[7] Copty, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Vardi, *Benefits of bounded model checking at an industrial setting*, in: *Computer Aided Verification, Proc. 13th International Conference*, Lecture Notes in Computer Science **2102** (2001), pp. 436–453.

[8] Kupferman, O. and M. Vardi, *Model checking of safety properties*, Formal methods in System Design **19** (2001), pp. 291–314.

[9] Manna, Z. and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems: Safety," Springer-Verlag, New York, 1995.

[10] McMillan, K., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

[11] Sajid, K. and R. Kaviola, *Verification of pentium$^{TM}$4 BUS recycle logic using symbolic simulation and induction*, in: *Intel Design Test and Technology Conference*, 2003.

[12] Seger, C. and R. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design **6** (1995), pp. 147–189.

[13] Sheeran, M., S. Singh and G. Stalmarck, *Check safety properties using induction and a SAT-solver*, in: *Proc. 3rd Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science **1954** (2000), pp. 108–125.

[14] Vardi, M. and P. Wolper, *Reasoning about infinite computations*, Information and Computation **115** (1994), pp. 1–37.

# Verifying Industrial Hybrid Systems with MathSAT [1]

Gilles Audemard [a], Marco Bozzano [b], Alessandro Cimatti [b] and Roberto Sebastiani [c,2]

[a] *Centre de Recherche en Informatique de Lens*
*IUT de Lens, Rue de l'université, SP16, F 62307 Lens Cedex*
audemard@iut-lens.univ-artois.fr

[b] *ITC-IRST, Via Sommarive 18, 38050 Povo, Trento, Italy*
{bozzano,cimatti}@itc.it

[c] *DIT, Università di Trento, Via Sommarive 14, 38050 Povo, Trento, Italy*
rseba@dit.unitn.it

**Abstract**

Industrial systems of practical relevance can be often characterized in terms of discrete control variables and real-valued physical variables, and can therefore be modeled as hybrid automata. Unfortunately, continuity of the physical behaviour over time, or triangular constraints, must often be assumed, which yield an undecidable class of hybrid automata.

In this paper, we propose a technique for bounded reachability of linear hybrid automata, based on the reduction of a bounded reachability problem to a MathSAT problem, i.e. satisfiability of a boolean combination of propositional variables and mathematical constraints. The MathSAT solver can be used to check the existence (or absence) of paths of bounded length.

The approach is very similar in spirit to SAT-based bounded model checking; furthermore, the ability to reason directly about real variables gives computational leverage over discretization-based methods. Despite the undecidability of the general problem, the proposed method is able to provide valuable information on large designs of practical relevance.

*Key words:* Formal Verification, Hybrid Systems, SAT

[1] This work has been sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. It has also been partly supported by ESACS, an European sponsored project, contract no. G4RD-CT-2000-00361, and by a grant from Intel Corporation.

[2] Sponsored by a MIUR COFIN02 project, code 2002097822_003.

*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

Many systems and plants of industrial relevance (e.g., engines, turbines) are defined in terms of discrete control variables and physical real-valued variables (e.g., speed, pressure), and can be naturally modeled as hybrid automata: depending on a discrete state (e.g., "nominal", "increasing"), different equations describe the behaviour of the physical variable (e.g., speed). Frequently, the dynamics of physical variables is continuous: i.e., transitions from a discrete state to another should not necessarily yield a discontinuity in the physical dimension. For instance, in the transition from "increasing" to "decreasing", the velocity should not change its value (but only its derivative). Furthermore, the evolution can depend on the comparison between the values of physical variables. Unfortunately, either imposing continuity or allowing for comparisons between variables (also known as triangular constraints) result in a class of hybrid automata where even reachability is undecidable [12]. Yet, it is very important to be able to develop tools that allow to formally validate such designs, that often implement critical functionalities (e.g., control systems for avionics).

In this paper, we address the problem of verifying hybrid automata with continuous variables and triangular constraints. We propose a formal verification method for bounded reachability. The approach is based on the encoding of a bounded reachability problem into a MATHSAT problem, i.e. the problem of checking the satisfiability of a boolean combination of propositional variables and mathematical constraints over real variables. The approach is made practical by the use of the efficient MATHSAT solver [1], that extends and integrates state-of-the-art techniques for propositional satisfiability (SAT) with a set of mathematical reasoners. The approach presented in this paper is largely similar to bounded model checking [4], and enhances the method presented in [3], limited to timed systems, to dealing with real variables with arbitrary linear dynamics.

The proposed technique is clearly incomplete, and currently limited to the case of linear dynamics. Despite these facts, however, it allows us to represent and to analyze interesting systems from real-world applications [6,5], providing useful information, especially oriented to debugging and goal-directed simulation. An experimental analysis shows that our techniques is competitive with state of the art verification tools such as HYTECH, and with methods based on the discretization of real variables.

### Outline of the paper

The rest of the paper is structured as follows. In Section 2 we illustrate a motivating example for our approach; in Section 3 we give a short and informal introduction to our model of hybrid systems; in Section 4 we give a brief overview of SAT-based bounded model checking and we discuss in more detail our encoding of hybrid systems into MATHSAT; in Section 5 we discuss

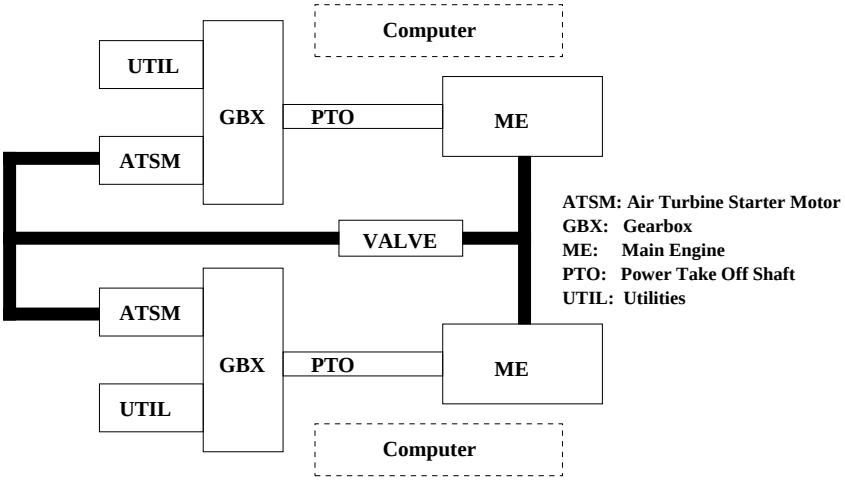


Fig. 1. SPS schematic view

some experiments, and finally in Section 6 and 7 we discuss related work and draw some conclusions.

## 2 A Motivating Example: The Secondary Power System

Throughout the paper, we use a running example to motivate and illustrate the main concepts we present. Specifically, we discuss the modeling and analysis of a real-world safety-critical system, namely the Secondary Power System (SPS). It is an industrial case study which has been and is being investigated within ESACS (Enhanced Safety Analysis for Complex Systems), a European-Union-sponsored project in the avionics sector, whose goal is to define a methodology to improve the safety analysis practice for complex systems development [6,5].

The SPS drives the hydraulic and electrical utilities of an aircraft. It is an example of safety-critical system with embedded hardware and software components. The hardware subsystems comprise (electro)-mechanical components (e.g., control valves, relays, shafts, gearboxes, freewheels) and electronic transducers (e.g., speed and pressure sensors), whereas the software component is given by embedded controllers (SPS computers).

The SPS drives the utilities of both the left and right hand side of the aircraft. To ensure the basic safety requirement, i.e. *no single failures shall cause the total loss of the SPS utilities*, the architecture of the system includes two basic redundancies: there are two independent and perfectly symmetric lines, whose purpose is to drive the left and the right hand side utilities, respectively; for each side, the mechanical drive of the relevant utilities (normal mode) is redounded by a pneumatic drive (*cross-bleed mode*) in case of failure of one of the components in the mechanical line.

Figure 1 shows a simplified schematic view of the SPS. The SPS normal

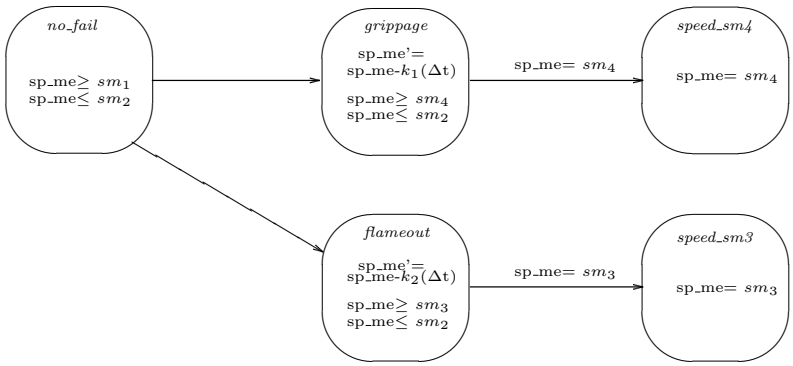


Fig. 2. SPS: main engine automaton (ME)

operation consists in transmitting the mechanical power from the engines to the relevant hydraulic and electrical generators. Specifically, the mechanical power of the main engine (ME) is transmitted via the Power Take Off Shaft (PTO) to a gearbox (GBX) which feeds the utilities. A component may fail due to abnormal operational conditions or ruptures. As an example, *flameout* and *grippage* are two possible failure modes of the main engine. To ensure safety of in-flight operation, in case of an engine failure the SPS computers automatically initiate a *cross-bleed* procedure consisting in driving the hydraulic and electrical generators by means of an air turbine motor (ATSM), using bled air from a valve (VALVE), which is in turn fed by the mechanical power coming from the opposite engine. Correct functioning of the cross-bleed procedure is an example of one safety requirement of the SPS. Some experimental results about this will be presented in Section 5.

## 3 Modeling Hybrid Automata

In this section we briefly present and exemplify our model of hybrid systems. The model is inspired by the *linear* and *rectangular hybrid automata* models presented in [10,11]. Informally, a hybrid system can be seen as the parallel composition of a collection of hybrid automata, which can communicate either by explicit synchronization on some *channel*, or implicitly by means of *shared variables*. Each automaton models both *discrete* events (e.g., failure of a component) and continuous activities of analog variables (e.g., time, component speed). At any given instant of time, the state of a hybrid automaton is defined by a control location (discrete state) and the values of all the analog variables (continuous state). The state can change either because of an *instantaneous* discrete transition, which changes the control location and may also affect the values of the analog variables (e.g., re-initialization is possible) or because of a *time elapse* (continuous) transition, which changes only the values of the analog variables according to some specified law. Hybrid systems can be seen as an extension of the timed systems model of [3], in which the only analog variables are clocks. In the following, by *elementary linear ex-*

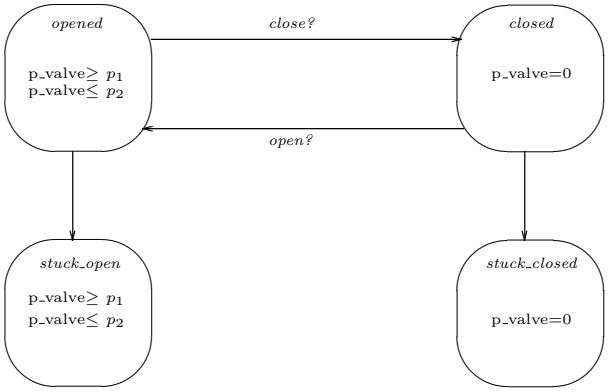


Fig. 3. SPS: valve automaton (VALVE)

*pression* we mean an equality and/or (non-strict) inequality over *linear terms* (i.e., linear combinations of real-valued variables with rational coefficients).

Figure 2 and 3 depict two examples of hybrid automata, modeling the *main engine* (ME) and the valve (VALVE) components of the SPS. A hybrid automaton consists of the following components:

**Locations** A finite set of locations, encoding the discrete states of the hybrid automaton. The automaton in Figure 2 has five locations, drawn as circles, which model the discrete state of the ME of the SPS. Location $no\_fail$ models the default behaviour of the engine; locations *grippage* and $flameout$ model two different faulty states; locations *speed_sm*4 and *speed_sm*3 model states in which the speed of the engine has the constant value $sm_4$ and $sm_3$.

**Analog Variables** A finite vector of real-valued data variables $(w_1, \ldots, w_n)$. The *sp_me* variable in Figure 2 encodes the speed of the ME. *Clock* variables of [3] may be seen as a particular case of real-valued variables. Primed variables (e.g., *sp_me*$'$) are used to denote the value of real-valued variables after execution of a transition.

**Initial and Invariant Conditions** Every location of a hybrid automaton may be declared as initial (meaning that it is a legal initial state of the system). Every location may be equipped with *invariants* on the real-valued variables, expressed by means of a set of elementary linear expressions $\{\psi_1, ..., \psi_h\}$ over the variables $w_1, \ldots, w_n$. Location $no\_fail$ is the initial location of the ME automaton (Figure 2), and is equipped with an invariant enforcing the *sp_me* variable to stay between the constant values $sm_1$ and $sm_2$. The invariant in location *speed_sm*3 forces *sp_me* to assume constantly the value $sm_3$.

**Channels** A finite set of channels is used for discrete communication between automata. A channel $c$ may be used as an *input* (notation $c?$) or an *output* (notation $c!$) channel for synchronizing different automata. For instance, the pressure valve automaton of Figure 3 uses two different input channels called *open*? and *close*?. The intended semantics is that the pressure valve

automaton awaits for incoming commands (requesting either opening of closing of the valve) coming from the relevant SPS computer controller.

**Transitions** A finite set of discrete transitions encodes the *discrete* evolution of the automaton. Each transition (also called *switch*) has a *source* and *target* location, and may be equipped with a set $\{\gamma_1, ..., \gamma_k\}$ of *guards* (pre-conditions) and a set $\{\theta_1, ..., \theta_m\}$ of *jump conditions* (post-conditions) on the real-valued variables. A guard is an elementary linear expression over $w_1, \ldots, w_n$; a jump condition is an elementary linear expression over $w_1, \ldots, w_n, w'_1, \ldots, w'_n$. In Figure 2, the transition from $flameout$ to $speed\_sm3$ has a guard $sp\_me = sm_3$ and no jump condition. By convention, the absence of jump conditions on a transition forces real-valued variables to preserve their value (e.g., in the previous example $sp\_me' = sp\_me$ implicitly holds). Transitions may be equipped with one or more optional labels denoting the channels on which the automaton must synchronize. For instance, two transitions in Figure 3 are labeled with the input channels *open*? and *close*?

**Variable Dynamics** Variable dynamics describe how the real-valued variables change in presence of a time elapse transition, and are expressed, for each location, as a set $\{\Psi_1, \ldots, \Psi_k\}$ of elementary linear expressions over $w_1, \ldots, w_n, w'_1, \ldots, w'_n$, As an example, in Figure 2 the *sp_me* variable in location *grippage* varies according to the law $sp\_me' = sp\_me - k_1(\Delta t)$ (where $k_1$ is a constant), that is, the speed decreases linearly (proportionally to the time delay) with first derivative equal to $k_1$. The expression $\Delta t$, encoding the time delay, will be explained in Section 4.2.

The hybrid automata presented here do not fall into the *rectangular* automata class described in [10], since re-initialization of variables is not enforced, and triangular constraints are possible. As a consequence, even the problem of reachability for this class of automata is undecidable [12].

## 4 Bounded Model Checking for Hybrid Systems

In this section we give a very short overview of SAT based model checking, and we discuss the encoding of our model of hybrid systems, informally described in Section 3, into MATHSAT.

### *4.1 SAT Based Bounded Model Checking*

Bounded Model Checking (BMC) is a recent approach to symbolic model checking [4]. Given a Kripke structure $M$, and an LTL formula $f$, the idea is to check whether $f$ is true in $M$ by looking for a counterexample (i.e., a witness to the violation of $f$) that can be presented within a bound of $k$ steps. Given $k$, the problem is reduced to the satisfiability of a propositional formula $[[M, \neg f]]_k$. For instance, for a property of the form $f := \mathbf{G}p(\underline{s})$, where $p(\underline{s})$ is

a boolean formula in the boolean variables $\underline{s}$, then

$$[[M, \neg f]]_k = \mathcal{I}(\underline{s}^{(0)}) \wedge \bigwedge_{i=0}^{k} \mathcal{C}(\underline{s}^{(i)}) \wedge \bigwedge_{i=0}^{k-1} \mathcal{R}(\underline{s}^{(i)}, \underline{s}^{(i+1)}) \wedge \bigvee_{i=0}^{k} \neg p(\underline{s}^{(i)}),$$

where $\mathcal{I}(\underline{s}^{(0)})$ is a representation of the initial conditions, $\mathcal{C}(\underline{s}^{(i)})$ is a representation of the invariant conditions at step $i$, and $\mathcal{R}(\underline{s}^{(i)}, \underline{s}^{(i+1)})$ is a representation of the transition relation from step $i$ to step $i+1$. If $[[M, \neg f]]_k$ is satisfiable, the propositional model provides a counterexaple of $k$ steps to $f$. If $[[M, \neg f]]_k$ is unsatisfiable, then nothing can be said about the existence of counterexamples to $M \models f$ with higher bound. The typical technique is to generate and solve $[[M, \neg f]]_k$ for increasing values of $k$, until either a counter-example is found, or a given time-out is reached.

BMC is being increasingly accepted as practical technique, effective in particular in the process of falsification, i.e. bug finding. The technique relies on the use of efficient SAT solvers (e.g., based on DPLL procedures [8]) for checking the propositional satisfiability of $[[M, \neg f]]_k$. As shown in [7], BMC avoids the blow-up in memory that can occur with model checking based on Binary Decision Diagrams, and is therefore able to tackle much larger circuits.

### *4.2 The encoding*

Our approach to the verification of hybrid automata is a generalization of BMC for timed systems, as proposed in [3]. The approach reduces a BMC problem for timed systems to the problem of deciding the satisfiability of math-formulas, i.e. boolean combinations of boolean variables and linear (in)equalities over real variables, representing absolute time and clocks. The resulting math-formulas are tackled with MATHSAT [2,1], a solver combining an efficient DPLL procedure with mathematical constraint solvers of increasing deductive power.

In the encoding for timed automata, boolean variables are used to encode the discrete part of the system, while linear constraints on real variables encode the timed part. In particular, each location $l$ is represented by a bitwise encoding $\underline{l}$, so that $\underline{l_i}$ holds if and only if the system is in the location $l_i$; each synchronization event (channel, shared variable) is represented by a corresponding boolean variables; each switch is represented by a single boolean variable (say, $T$) which holds if and only if the system executes the corresponding switch; a boolean variable $T_\delta$, representing a continuous transition, holds if and only if time elapses by some $\delta > 0$; finally, for each process $P_i$, we introduce a boolean variable $T^i_{null}$, that holds if and only if process $P_i$ does nothing. In order to deal with time, we introduce a real valued variable $t$ representing absolute time, and, for each clock $x$, a real valued variable $ox$ representing the difference with respect to absolute time. All mathematical constraints required in the encoding are in the form $v_1 - v_2 \bowtie c$, $\bowtie \in \{\leq, \geq, =, >, <\}$ $v_1$

and $v_2$ being real variables representing either absolute time or clock values. The reader can refer to [3] for details.

We tackle the case of hybrid automata by considering that it is an extension of the case of timed automata. The encoding for the timed case is extended by introducing a set of real variables $\omega_i$'s, representing physical entities. To simplify the notation, in the following we write: "$\Delta t$" for the difference $t' - t$ between absolute time in the next and in the current state; "$\Delta\omega$" for "$\omega' - \omega$", so that, e.g., we write "$c \cdot \Delta t ...$" for "$c \cdot t' - c \cdot t ...$"; "$\Delta\omega = 0$" for "$\omega' = \omega$", "$\Delta\omega \leq ...$" for "$\omega' \leq \omega + ...$". We also write "$(w \in [t_1, t_2])$" for "$(w \geq t_1) \wedge (w \leq t_2)$", where $t_1$ and $t_2$ are linear terms. If $\psi$ is a formula, $\psi'$ denotes the formula obtained by substituting in $\psi$ each propositional variable $p_j$ with $p'_j$, and each real variable $v_i$ with $v'_i$.

*4.2.1 Initial conditions $\mathcal{I}(\underline{s}^{(0)})$.*
At step 0, in an initial location $l$, $\omega$ can either:

- be set to a given initial value $c_0$. If so, we represent this fact by the axiom:

$$\underline{l}^{(0)} \rightarrow (\omega^{(0)} = c_0); \tag{1}$$

- be set nondeterministically to an initial value within a closed interval $[a_0, b_0]$, $a_0, b_0 \in [-\infty, \infty]$. [3] If so, we represent this fact by the axiom:

$$\underline{l}^{(0)} \rightarrow (\omega^{(0)} \in [a_0, b_0]). \tag{2}$$

*4.2.2 Invariant conditions $\mathcal{C}(\underline{s})$.*
For each location $l$ equipped with the set $\{\psi_1, ..., \psi_h\}$ of invariants on real valued variables, we include the axiom

$$\underline{l} \rightarrow \bigwedge_j \psi_j. \tag{3}$$

*4.2.3 Transition relation $\mathcal{R}(\underline{s}, \underline{s}')$.*
For each switch $T$ equipped with a set $\{\gamma_1, ..., \gamma_k\}$ of guards and with a set $\{\theta_1, ..., \theta_m\}$ of jump conditions on the real valued variables $\omega_i$'s and $t$, we include the axioms

$$T \rightarrow \bigwedge_j \gamma_j, \tag{4}$$

$$T \rightarrow \bigwedge_j \theta'_j \tag{5}$$

[3] "$a_0 = -\infty$" and "$b_0 = -\infty$" mean that there is no lower bound and no upper bound for $\omega$ respectively.

respectively. For each physical variable $\omega$ that is not interested by a jump condition of switch $T$, and must therefore keep its value, we add the axiom:

$$T \rightarrow (\Delta\omega = 0). \tag{6}$$

When process $i$ does nothing, in correspondence of $T^i_{null}$, each physical variable $\omega$ maintains its value:

$$T^i_{null} \rightarrow (\Delta\omega = 0). \tag{7}$$

When time elapses in a location $l$, physical variables $\omega_i$ evolve according to the set of variable dynamics $\{\Psi_1, \ldots, \Psi_k\}$ associated with $l$. For each location, we add the axiom

$$(\underline{l} \wedge T_\delta) \rightarrow \bigwedge_i \Psi_i \tag{8}$$

Different forms of variable dynamics are possible:

- $\omega$ maintains its value under a dynamic of the form:

$$(\underline{l} \wedge T_\delta) \rightarrow (\Delta\omega = 0); \tag{9}$$

- $\omega$ may evolve deterministically according to a linear function:

$$(\underline{l} \wedge T_\delta) \rightarrow (\Delta\omega = c \cdot \Delta t) \tag{10}$$

  $c$ being a constant.

- $\omega$ may evolve nondeterministically within two linear functions:

$$\omega' \in [b_1\omega + c_1 \cdot \Delta t - a_1, b_2\omega + c_2 \cdot \Delta t + a_2], \tag{11}$$

$$a_1, a_2 \geq 0,\ b_1, b_2 \in \{0, 1\},\ c_1, c_2 \in (-\infty, \infty). \tag{12}$$

  If $a_1 = a_2 = 0$, then (11) encodes a triangular constraint. If $b_1 = b_2 = 0$ and $c_1 = c_2 = 0$, then (11) encodes a rectangular constraint.

- in the general case, the evolution of the variables can be nondeterministic within the space described by the linear inequalities $\{\Psi_1, \ldots, \Psi_k\}$, as in equation 8.

The encoding of properties basically follows the encoding in [3]. Our approach is bounded complete, in the following sense: if there exists a trace of length $k$, then the encoding of length $k$ is satisfiable, and can be found by running MATHSAT on it. The undecidability of the class of hybrid automata we are dealing with tells us that it is in general impossible to decide if a counterexample might be found with bigger $k$, or if the problem is unsolvable.

## 5 Experimental Evaluation

We evaluated the potential of the approach by tackling an example of hybrid systems of industrial relevance, i.e. the model of the SPS. The bounded reachability method described in Section 4 can be used both for model debugging

**Time** $T_0$ :

**Locations** : *no_fail*, *gbx_pto_driven*, *atsm_idle*, *sps_ok*, *closed*
**Analog Variables** : $sp\_me = sm_2$, $sp\_gbx = sg_2$, $sp\_atsm = 0$
**Discrete Trans** : *me_grippage*
**Synchronizations** : none

**Time** $T_1$ :

**Locations** : *grippage*, *gbx_pto_driven*, *atsm_idle*, *sps_ok*, *closed*
**Analog Variables** : $sp\_me = sm_5$, $sp\_gbx = sg_3$, $sp\_atsm = 0$
**Discrete Trans** : *atsm_inc_a*, *sps_inc_a*, *valve_open*
**Synchronizations** : SPS and ATSM on *inc_a*, SPS and VALVE on *open*

**Time** $T_2$ :

**Locations** : *grippage*, *gbx_pto_driven*, *atsm_inc_a*, *sps_inc_a*, *open*
**Analog Variables** : $sp\_me = sm_6$, $sp\_gbx = sg_4$, $sp\_atsm = sa_2$
**Discrete Trans** : *atsm_inc_a_inc_b*, *sps_inc_a_inc_b*
**Synchronizations** : SPS and ATSM on *inc_b*

**Time** $T_3$ :

**Locations** : *grippage*, *gbx_pto_driven*, *atsm_inc_b*, *sps_inc_b*, *open*
**Analog Variables** : $sp\_me = sm_7$, $sp\_gbx = sg_1$, $sp\_atsm = sa_3$

Fig. 4. An example of MathSAT trace

(i.e., bug hunting) and for simulation of hybrid systems. In the following we provide some hints about the use of our methodology by showing some experimental results. For illustration purposes, we will discuss a simplified *one-sided* model of the SPS case study, including one instance of the ME, GBX, VALVE, ATSM, PTO and SPS computer components of Figure 1. Under this abstraction, the analogous components of the opposite side of the system are assumed to be correctly working. An example of property to be checked is given by (the negation of) the following formula:

$$(\texttt{!GBX.loc\_broken} \;\&\; \texttt{!GBX.loc\_grippage} \;\&\; \texttt{!VALVE.loc\_stuck\_closed} \;\&\; \texttt{!ATSM.loc\_broken} \;\&\; \texttt{!PTO.loc\_fused}) \;\; \texttt{U} \;\; \texttt{GBX.sp\_gbx} <= \texttt{sg}_1 \qquad \text{(P1)}$$

This is a typical safety property expressed via the LTL *until* operator. The intended semantics is whether there exists a path such that no failures of the GBX, VALVE, ATSM and PTO components happen along the path, and finally the speed of the *gearbox* (GBX component) drops below the constant value $sg_1$. The negation of the above property can be seen as a *safety* property to be verified by the system (i.e., *in presence of failures due only to the main engine, the gearbox speed cannot drop below* $sg_1$). The rationale behind this property is that the *cross-bleed* procedure initiated by the SPS computer (see Section 2) is able to recover from an engine failure by using power coming from the opposite engine (which is assumed to be working in this one-sided model).

The property may or may not hold depending on the value chosen for the constant $sg_1$. In particular, if the value chosen for $sg_1$ exceeds a given threshold, the property is falsified by MathSAT (this means that the cross-bleed procedure is not able to prevent the gearbox speed to drop below that particular value). In this case, MathSAT generates an output trace showing an execution of the system which leads to the violation. The trace includes information on the discrete transitions and the time elapse transitions taken by the automata, the exact time delays and time points at which the transitions take place, and the synchronization channels between different automata. If the value of the constant $sg_1$ is chosen below a suitable threshold, property (P1) holds, and therefore MathSAT correctly does not find any counterexample. Regarding the choice of the constant $sg_1$, see the discussion in Section 7.

The trace generated by MathSAT is schematically shown in Figure 4. For each time instant, the trace shows the current locations of the ME, GBX, ATSM, and VALVE automata, the current values of the *sp_me*, *sp_gbx*, *sp_atsm* analog variables, the discrete transitions which take place at that time instant, and the synchronizations channels. For a better understanding of the trace, in Figure 5 we show a simplified version of the SPS computer automaton (only the part relevant to the simulation is shown). Notice that this automaton shows as example of triangular constraint, i.e. $sp\_gbx - sp\_atsm \leq c_1$ $[\geq c_1]$, and of communication with shared variables (variables *sp_gbx* and *sp_atsm* model, respectively, the speed of the GBX and ATSM components).

The simulation begins at time T0, when an engine grippage takes place. Both the engine and the gearbox speeds begin to decrease. At time T1, the SPS computer detects a gearbox low speed condition, and therefore issues the opening of the valve (the VALVE and SPS computer automata synchronize on the *open* channel); as a result, the ATSM begins to increase its speed (SPS and ATSM synchronize on the *inc_a* channel). At time T2, the SPS computer issues a change in the ATSM dynamics (SPS and ATSM synchronize on *inc_b*). The simulation stops at time T3, when the gearbox speed reaches the value $sg_1$.

The same approach can be used for guided simulation. To give an example, we consider the following formula:

$$\begin{aligned}&(\texttt{!ME.loc\_eng\_flameout} \;\&\; \texttt{!GBX.loc\_broken} \;\&\; \texttt{!GBX.loc\_grippage} \;\&\\ &\texttt{!VALVE.loc\_stuck\_closed} \;\&\; \texttt{!ATSM.loc\_broken} \;\&\; \texttt{!PTO.loc\_fused})\\ &\quad \texttt{U} \;\; \texttt{GBX.sp\_atsm} >= \texttt{sa}_1 \end{aligned} \tag{P2}$$

It is a variation of the previous reachability property, here we require that at the end of the path the speed of the ATSM component is greater than the constant value $sa_1$. Furthermore, by explicitly ruling out an engine *flameout*, we limit the possible failure modes of the main engine to *grippage*. As explained in Section 2, in presence of an engine failure, the ATSM component is responsible of carrying out the *cross-bleed* procedure, which consists in driv-

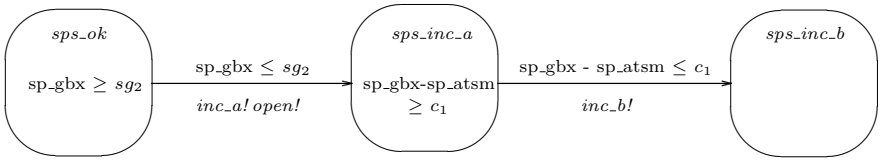


Fig. 5. SPS computer automaton (fragment)

| | Property P1 | | | | Property P2 | | | |
|---|---|---|---|---|---|---|---|---|
| $k$ | Time | Σ Time | Mem. | Result | Time | Σ Time | Mem. | Result |
| 2 | 0.06 | 0.06 | 5.6 | UNSAT | 0.10 | 0.10 | 5.5 | UNSAT |
| 3 | 0.20 | 0.26 | 6.2 | UNSAT | 0.16 | 0.26 | 6.0 | UNSAT |
| 4 | 0.53 | 0.79 | 7.1 | UNSAT | 0.30 | 0.56 | 6.8 | UNSAT |
| 5 | 1.81 | 2.60 | 7.7 | UNSAT | 0.49 | 1.05 | 7.5 | UNSAT |
| 6 | 6.49 | 9.09 | 8.4 | UNSAT | 0.84 | 1.89 | 8.2 | UNSAT |
| 7 | 4.53 | 13.62 | 8.9 | SAT | 1.53 | 3.42 | 8.8 | UNSAT |
| 8 | | | | | 2.88 | 6.30 | 9.4 | UNSAT |
| 9 | | | | | 4.94 | 11.24 | 10.0 | UNSAT |
| 10 | | | | | 8.69 | 19.93 | 10.7 | UNSAT |
| 11 | | | | | 8.88 | 28.81 | 11.3 | SAT |

Table 1
Experimental results (Time in seconds, Memory in MB)

ing the gearbox with the pneumatic power coming from the valve. Correct functioning of the cross-bleed procedure requires the ATSM (which is initially idle) to start and bring up the gearbox speed. Using MathSAT, we are able to reconstruct a trace corresponding to the above property, which illustrates how the cross-bleed procedure is carried out. It is possible to tune the above simulation and perform further ones by adding further constraints on the trace to look for.

The performance of our method on the examples described above are reported in Table 1. For each problem length, we show computation time, total computation time up to that problem instance, and memory usage. Computation times include both parse and search time. The results have been obtained on a Pentium III machine 1.0 GHz, with 256 Mb memory, running Linux Redhat 7.1. The minimal length trace generated by MathSat for P1 has length 7, whereas the one generated for P2 has length 11.

We also attempted a comparison with HYTECH [11], a state-of-the-art tool for the analysis of hybrid systems. Differently from our approach, HYTECH is based on the calculation of the reachable state space, and is therefore not limited to the bounded case. In principle, HYTECH may not terminate when tackling an undecidable class of automata (as in the case of the SPS).

We encoded the models of the SPS, as closely as possible, into HYTECH. Overflow errors in the underlying polyhedral libraries made it impossible for HYTECH to compute the space of reachable configurations beyond the 5th iteration. We also attempted to use the `-o1` and `-o2` options, that are sometimes able to limit such problems, but in our case obtained no effect. From the

point of view of performance, the time required by HYTECH to reach the 5th iteration was 32 seconds, when run without options; the use of `-o1` and `-o2` required 50 and 86 seconds, respectively. The analysis is very preliminary, but seems to suggest that there is a clear potential in our techniques.

## 6 Related Work

The work presented in this paper builds upon our previous work on *timed systems* [3]. In [3], we showed how to reduce the problem of bounded model checking for timed systems to the satisfiability of a math-formula, which can then be checked by a SAT-solver. We also presented the MathSat solver [2,1], an efficient SAT-solver which is based on the integration of SAT techniques [4] with some specialized decision procedures for linear mathematical constraints. A work related to ours, but still limited to timed systems, is [17]. In the present work, as explained in Section 4.2, we have extended the encoding in order to deal with hybrid systems.

Our model for hybrid systems is closely related with the linear and rectangular hybrid automata models presented in [10,13], the main difference being in the definition of the dynamics of the real-valued variables. In [10], the dynamics (called *flow conditions*) of the real-valued variables are defined by means of linear constraints over the first derivatives of such variables, whereas in our model dynamics can be characterized by means of linear functions of the time delay, which directly constrain the behaviour of the variables. This approach is analogous to restricting the flows of the real-valued variables to stay inside a rectangular region, as in the rectangular automata model of [10]. In fact, as noted in [12], under the hypothesis of working with *convex* linear constraints, requiring the flow to be inside a rectangular region amounts to requiring the existence of a smooth function inside the corresponding piecewise-linear envelope.

The model of hybrid I/O automata presented in [14] is general enough to accommodate our model of hybrid automata. Discrete and continuous communication are achieved by means of, respectively, shared actions and shared variables. However, discrete events are not allowed to change the value of shared variables, as in our case.

As an alternative approach to the verification of hybrid systems, we cite [15], where the CHECKMATE tool is presented. CHECKMATE performs verification of hybrid systems using finite-state approximations called *quotient transition systems*. Although this approach is not restricted to linear hybrid automata, the verification analysis may be inconclusive, in which case a refinement of the current approximation may be attempted. An analysis of the current trends in model checking of hybrid systems can be found in [16].

This line of research has been carried on inside the ESACS [6] project (see `http://www.esacs.org`), an European-Union-sponsored project whose main goals are to define a methodology and a shared environment to improve

the safety analysis practice for complex systems development. The Secondary Power System [5] is one of the case-studies investigated in ESACS. One of the main motivations for our research is the realization that the use of traditional finite-state model checking, based on the discretization of real variables, has a very hard time in dealing with the complexity of hybrid systems [5]. In fact, the results may depend on the step of discretization, and the state explosion problem makes such an approach infeasible in practice.

## 7 Conclusions and Future Work

In this paper, we have addressed the problem of verification of industrial systems that are naturally modeled as linear hybrid automata. The approach is an enhancement of the bounded model checking approach for timed systems proposed in [3] to the case of linear hybrid automata. Efficiency is gained by the use of the MathSAT solver. The main limitations are given by the undecidability of the analyzed class, and the constraints on the linearity of real variables dynamics. Despite this, however, the approach allows us to model and analyze systems of practical relevance, that HYTECH is currently unable to deal with.

In the future, we will provide a more thorough experimental evaluation, by enlarging both the set of tools we compare with (some of them are cited in Section 6), and the set of case studies to analyze. Regarding the SPS example, we plan to experiment with more complex models, at different levels of granularity and abstraction (e.g., a two-sided model of the system). We will investigate how to optimize the MathSAT solver on these specific problems (e.g., by constraining the splitting variables in the style of [9,18]), and will experiment with different encodings. As a first step towards bridging the gap between *bounded* model checking and *unbounded* verification, inductive reasoning techniques to prove invariant properties will be investigated. An important point we plan to address in the near future is concerned with *parametric* analysis, which is currently supported in HYTECH. To exemplify, parametric analysis would allow us to replace the constant value $sg_1$ in property (P1) (see Section 5) with a parameter $\alpha$ in order to find out constraints on the parameter for which the property does or does not hold. Finally, in the future we plan to extend the framework to properties expressed in full LTL.

## References

[1] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Andrei Voronkov, editor, *CADE-18: Conference on Automated Deduction*, volume 2392 of *LNAI*, pages 195–210, Copenhagen, Denmark, 2002. Springer.

[2] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In Jacques Calmet, Bernard Benhamou, Olga Caprotti, Laurent Henocque, and Volker Sorge, editors, *CALCULEMUS-2002: Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, volume 2385 of *LNAI*, pages 231–245, Marseille, France, 2002. Springer.

[3] Gilles Audemard, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Bounded Model Checking for Timed Systems. In Doron A. Peled and Moshe Y. Vardi, editors, *FORTE 2002: Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *LNCS*, Houston, Texas, November 2002. Springer.

[4] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *Proc. 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[5] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving Safety Assessment of Complex Systems: An industrial case study. In *Proc. Formal Methods Europe (FME 2003)*, volume 2805 of *LNCS*, pages 208–222, 2003.

[6] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Lüdtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, and G. Zacco. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *European Safety and Reliability Conference (ESREL'03)*, pages 237–245. Balkema Publisher, 2003.

[7] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. CAV'2001*, LNCS. Springer, 2001.

[8] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[9] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Proc. AAAI'98*, pages 948–953, 1998.

[10] T.A. Henzinger. The Theory of Hybrid Automata. In *Proceedings 11th Annual International Symposium on Logic in Computer Science (LICS'96)*, pages 278–292, New Brunswick, New Jersey, 1996. IEEE Computer Society Press.

[11] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[12] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's Decidable About Hybrid Automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[13] T.A. Henzinger and R. Majumdar. Symbolic Model Checking for Rectangular Hybrid Systems. In S. Graf and M. I. Schwartzbach, editors, *Proceedings 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 142–156, Berlin, Germany, 2000. Springer-Verlag.

[14] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O Automata. *Information and Computation*, 2003. To appear.

[15] B.I. Silva, K. Richeson, B.H. Krogh, and A. Chutinan. Modeling and verification of hybrid dynamical system using CheckMate. In *Proc. ADPM 2000, Automation of mixed processes: Hybrid Dynamic Systems*. Shaker Verlag, 2000.

[16] B.I. Silva, O. Stursberg, B.H. Krogh, and S. Engell. An assessment of the Current Status of Algorithmic Approaches to the Verification of Hybrid Systems. In *Proc. 40th Conference on Decision and Control*, 2001.

[17] M. Sorea. Bounded Model Checking for Timed Automata. In *Proceedings 3rd Workshop on Models for Time-Critical Systems (MTCS'02)*, Brno, Czech Republic, 2002.

[18] Ofer Strichman. Tuning SAT checkers for bounded model checking. In *Proc. CAV00*, volume 1855 of *LNCS*, pages 480–494, Berlin, 2000. Springer.

# Exploiting Target Enlargement and Dynamic Abstraction within Mixed BDD and SAT Invariant Checking

Gabriel P. Bischoff, Karl S. Brace

*Massachusetts Microprocessor Design Center, Intel Architecture's Group, Shrewsbury, MA*

G. Cabodi, S. Nocco, S. Quer [1]

*Dip. di Automatica e Informatica, Politecnico di Torino, Turin, Italy*

**Abstract**

In this paper, we propose a methodology to make Binary Decision Diagrams (BDDs) and Boolean Satisfiability (SAT) Solvers cooperate. The underlying idea is simple: We start a verification task with BDDs, we go on with them as long as the problem remains of manageable size, then we switch to SAT, without losing the work done on the BDD domain.

We propose *target enlargement* as an attempt to bring some of the advantages of state set manipulation from BDDs to SAT-based verification. We first, "enlarge" initial and target state sets of a given verification problem by affordable BDD manipulations. This step is carried on with a few breadth-first steps of traversal, or with what we call *high-density dynamic abstraction*, i.e., a new technique to collect under-approximate reachable state sets. Then, we perform SAT-based verification with the newly computed "enlarged" sets.

We experimentally test our methodology within an industrial environment, the Intel *BOolean VErifier* BOVE. Preliminary results on standard benchmarks (the ISCAS'89, ISCAS'89–addendum, and VIS suites), and industrial ones (the IBM Formal Verification Benchmark Library) are provided. Results show interesting improvements over state-of-the-art techniques: We could decrease CPU time up to a 5x factor, when performing verification with the same depth, or we could increase the verification depth up to 30%, when performing verification within the same time limit.

*Key words:* Invariant Checking, Satisfiability Solver (SAT), Binary Decision Diagrams (BDDs), Target Enlargement.

*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

Current design methods are so complex that simulation is no longer adequate. In current design frameworks, hundreds to thousands of bugs must be found and removed during the initial phases of the design. While finding bugs remains an important goal, it is also essential to be able to prove their absence, i.e., formally verifying the correctness, before starting the production process.

Binary Decision Diagrams [7] (BDDs) have been widely used to formally verify correctness because of their ability to exhaustively analyze a problem. Nevertheless, BDDs have never been able to deal with the largest models and problem instances. Boolean Satisfiability [25] (SAT) Solvers, on the other hand, have been gaining ground especially for their debugging capability adopting Bounded Model Checking [3,11] (BMC). Unbounded inductive verification [23] guarantees full verification, but it is more complex than BMC and typically converges at higher sequential depths than BDDs.

In this work, we explore a new way to make BDD-based and SAT-based tools cooperate. Our target is to trade-off space and time, i.e., to improve "time efficiency" of SAT-based verification with the help of affordable, i.e., "space manageable", BDD-based operations. First of all, we "enlarge" initial and target state sets of the given verification problem by means of BDD manipulations. Then we perform SAT-based verification exploiting the newly computed "enlarged" set. In this way, we partition a verification task between two different tools. Initial BDD operations are useful for their breadth-first state space visit, and for their ability to represent state sets. Whenever we switch to SAT, BDD-based state sets provide a tighter space pruning and an enforced convergence within the SAT engine. Our main contribution is to study the impact of enlarged sets over SAT-based (both bounded and unbounded) model checking. We show that our approach is a task partitioning strategy, a way to restrict the SAT search, and to tighten the convergence of the unbounded algorithm. Another contribution of our work is to propose a BDD sub-setting procedure mixing the high density paradigm with an effort to reduce the number of variables (abstraction) in the support of a BDD. We call this technique "dynamic abstraction".

Preliminary results are reported on standard benchmarks (the ISCAS'89, ISCAS'89–addendum, and VIS suites), and industrial ones (the IBM Formal Verification Benchmark Library [18]). They show interesting improvements in terms of both efficiency and power, and demonstrate how target enlargement is able to boost BMC and induction toward larger verification tasks.

The remainder of this paper is organized as follows. In Section 2, we introduce some preliminary concepts on notation, SAT problems and reachability analysis. Section 3 is dedicated to the related work. Section 4 introduces the outline of our approach justifying and describing algorithms for target enlargements within SAT. Section 5 describes our dynamic under-approximate

[1] Email: stefano.quer@polito.it

reachability analysis to compute target enlargements. Section 6 presents our top-level algorithm with some more implementation-level details. Section 7 presents our experimental evidence. Finally, Section 8 concludes the paper with a brief summary, and some possible future works.

## 2 Background

### *2.1 Model and Notation*

The sequential systems we address are usually modeled as Finite State Machines (FSMs). Each FSM is described by a Transition Relation $\mathsf{TR}(s, y)$, which indicates its present–next state behavior, and an initial state set $\mathsf{S}$.

An invariant property [2] $\mathsf{P}$ is checked by attempting to prove (or disprove) the reachability of its complement $\mathsf{T}$ (target state set, $\mathsf{T} = \neg\mathsf{P}$) from $\mathsf{S}$. For sake of simplicity, we will always refer to the above kind of properties, even though our techniques can be applied to a broader set, the LTL properties supported by SAT-based verification algorithms.

### *2.2 SAT-Based Bounded Model Checking and Inductive Verification*

In order to decide if a Boolean formula $f$ is satisfied, most solvers adopt variants of the basic Davis-Putnam recursive algorithm. SAT solvers generally operate on problems for which the propositional formula $f$ is specified in *Conjunctive Normal Form* (CNF). This form is a two-level decomposition: The logical AND of one or more *clauses*, each of which consists of the logical OR of one or more *literals*.

SAT-based BMC considers only paths of bounded length $k$ and builds a propositional formula $f$ that is satisfiable *iff* there is a counter-example (a path from $\mathsf{S}$ to $\mathsf{T}$) of the same length. Complete verification is usually achieved by BMC with longer and longer bounds or by inductive techniques [23,19]. In short, in inductive verification, a sequence of BMC steps with increasing bound is complemented with SAT-based induction checks. In [23] the authors call a *path* a sequence of states through $\mathsf{TR}$:

$$path(s_{[0..n]}) = \bigwedge_{0 \le i < n} \mathsf{TR}(s_i, s_{i+1}) \tag{1}$$

and they define *loopFree* a path that visits a state at most once:

$$loopFree(s_{[0..n]}) = path(s_{[0..n]}) \wedge \bigwedge_{0 \le i < j \le n} (s_i \neq s_j) \tag{2}$$

The property is proved correct, i.e., $\mathsf{S}$ and $\mathsf{T}$ are mutually unreachable, if the following conditions hold:

- *Base case*: No bounded path of length less than $k$ connects $\mathsf{S}$ to $\mathsf{T}$.

[2] Or AG CTL property.

```
FwdMC (TR, S, T)
    R_0 = S
    i = 0
    do
        if ((R_i ∧ T) ≠ ∅)
            return (Trace (TR, S, R, T))
        i = i + 1
        R_i = S ∨ Img (TR, R_{i-1})
    while (R_i ≠ R_{i-1})
    return (Pass)
```

Fig. 1. Forward Model Checking.

- *Inductive proof*: No simple path of length $k$ exists such that its first state is initial and no other state is initial, or such that its final state is in $\mathsf{T}$ and no other state is in $\mathsf{T}$.

The authors of [23] demonstrate that the resulting method is sound and complete. Intuitively, the two conditions introduced as inductive proof correspond to the fact that, starting from the initial state, the full reachable state space has been visited, or that, after the property has been proved correct for $k-1$ time steps, it cannot become false (i.e., starting from the target state, we cannot reach the initial state). For this reason, in the sequel we call the two inductive checks as "forward" or "backward" induction.

### *2.3 BDD-Based Model Checking*

Standard BDD-based *forward model checking* is presented in Figure 1. The procedure is based on the iterated application of the post-image (Img) function, to compute symbolic post-images of the set of state $\mathsf{R}_{i-1}$. $\mathsf{R}_i$ are the state sets generated at each traversal iteration. Notice that in the pseudo-code the whole reached state set is given as input to the image procedure, whereas any state set in the interval between the *newly* reached states and the *whole* reached states set could be used. On-the-fly tests for intersection with the target are done at each iteration, thus avoiding full computation of reachable states whenever $\mathsf{T}$ is reached before the fix-point. A counter-example is possibly computed, by the function Trace, starting from $\mathsf{S}$, $\mathsf{T}$ and the array $\mathsf{R}$ of frontier sets $\mathsf{R}_i$ identifying all admissible paths.

CTL model checking procedures are often implemented (as well as our exact search) as backward traversal procedures, so let us also mention here that an invariant can be verified by proving/disproving the mutual reachability of $\mathsf{S}$ and $\mathsf{T}$ in the backward direction. This is easily expressed by swapping the $\mathsf{S}$ and $\mathsf{T}$ sets, and changing the Img function with the PreImg computation in Figure 1. In the sequel we will call $\mathsf{FR}$ and $\mathsf{BR}$ the forward and backward reachable state set respectively.

*Approximate Traversals* [10,13] are a popular way to extend the applicability of reachability analysis to larger circuits. The approach is based on

the *approximate image* ($\text{Img}^+$) operator, returning over-estimations of exact images:

$$\text{Img}^+(\mathsf{TR}, \mathsf{FR_i}) \supseteq \text{Img}(\mathsf{TR}, \mathsf{FR_i}) \tag{3}$$

Notice that, in the sequel, we will indicate with $\mathsf{FR}^+$ and $\mathsf{BR}^+$ ($\mathsf{FR}^-$ and $\mathsf{BR}^-$) the over (under) estimations of the forward and backward reachable state set respectively.

## 3 Related Works and Comparison Remarks

With the advent of SAT-based BMC tools a lot of researchers compared SAT-based methods with more traditional BDD-based ones. As different researchers agree that the two approaches are essentially complementary, a lot of recent works concentrate on dovetailing the two approaches in a loose or strict fashion. In this section, we review, among these works, the ones more strictly related to our approach.

Gupta et al. [16,17] perform BDD-based reachability analysis by using a SAT procedure within symbolic image computation. They call their approach *BDDs at SAT Leaves*. More specifically, they use BDDs to represent state sets and a CNF formula to represent the transition relation. Symbolic image of a state set is computed by exhaustive SAT search of all solutions within the space of primary input, present and next state variables. However, rather than using SAT to enumerate each solution all the way down to a leaf, image switches to BDD-based computations at certain intermediate points within the SAT decision tree. This is done as a trade off between space complexity of BDDs and time complexity of full SAT enumeration. In a sense, this approach can be regarded as SAT providing a disjunctive decomposition for image computation into many sub-problems, each of which is handled symbolically using BDDs.

As far as SAT performance is improved by BDD learning, Cabodi et al. [9] propose BDD pre-processing by means of over-approximate reachability. The authors show how to translate over-approximate state sets from BDDs to CNF clauses, to be used by a SAT solver as an extra learning, which is redundant, yet able to improve overall performance in BMC.

Gupta et al. [14] propose an approach sharing similarities with the previous one. They start from the CNF representation of the problem, and they build BDDs of selected "structural" points to learn useful information, in order to improve the learning ability of the solver.

Another work by Gupta et al. [15] may be considered as a variation of [9] for the BMC case, as they also use over-approximate reachability analysis to constrain the BMC search. A novel idea in their work is the extension to induction-based unbounded verification, where the authors exploit over-approximate information as an additional (non redundant) constraint.

Our work shares with the previous ones a few guiding ideas. We use both BDD and SAT-tools to cope with their contrasting limits but we partition

the work-load in a new way. Moreover, in our approach, BDDs are not used to perform a learning activity virtually useful only to prune the subsequent search, but are adopted to partially truly perform the verification task. As in other approaches, we rely on a loose coupling between the BDD and the SAT tool, but we strongly interact with the model checker.

## 4 Methodology's Outline

In this section we overview our methodology. The approach we propose can be viewed:

- As a way to partition a verification task between a BDD and a SAT engine. We perform a preliminary effort with BDDs, we conclude the task through a SAT solver, working on the solution space left uncovered by BDD preprocessing. In other words, counterexamples are computed (or refuted) partially within the BDD domain and partially within the SAT one.
- As an optimization of SAT-based model checking, where the results of BDD preprocessing are used not merely to reduce the search area, but to further optimize SAT search in its target sub-space. More in detail, state sets generated during the BDD phase are used as a stronger constraint for the relative SAT problem, and to enforce the convergence of SAT-based unbounded model checking.

In the rest of this section we first show how BDD-based target enlargement can be considered in terms of task partitioning between the BDD and SAT tools. Then we introduce some specific optimizations for SAT searches. BDD preprocessing for target enlargement and the overall verification algorithm will be described in the following sections.

### *4.1 Target Enlargement as a Task Partitioning Strategy*

Target Enlargement [24,2] is a known paradigm in hardware verification. It is an effort to better coordinate and balance a verification workload between two tools or two different search procedures within the same tool.

Whenever a verification task looks for a path to a given target state, one might "enlarge" the target by computing a set of states reachable in the opposite direction from the target. Roughly speaking, the target is now replaced by a wider area, and the chance to reach it (or to prove it is unreachable) is now higher. Analogous considerations hold for the initial set of states.

We apply it by means of under-approximate BDD-based reachability starting either/both from $\mathsf{S}$ or/and $\mathsf{T}$. Our idea is to first compute BDD-based reachable state sets, so that they can be used as new targets for easier SAT processing.

Let us work on a SAT-based model checking problem where the goal is to prove the mutual reachability between $\mathsf{S}$ and $\mathsf{T}$. We can generate a related

SAT problem with new enlarged initial and/or target state sets $\mathsf{S}^e$ and $\mathsf{T}^e$, such that

$$\begin{aligned} \mathsf{S}^e &= \mathsf{FR}^-(\mathsf{S}) \subseteq \mathsf{FR}(\mathsf{S}) \\ \mathsf{T}^e &= \mathsf{BR}^-(\mathsf{T}) \subseteq \mathsf{BR}(\mathsf{T}) \end{aligned} \tag{4}$$

The new *enlarged* start (target) state set is a set of states for which a path from $\mathsf{S}$ (to $\mathsf{T}$) exists. BDD-based computation of $\mathsf{S}^e$ and $\mathsf{T}^e$ will be discussed in the next section. Let us just consider here the straightforward case of under-approximation by bounded exact reachability, i.e., a few traversal iterations, with $\mathsf{S}^e = \mathsf{FR}_{d_F}$ and $\mathsf{T}^e = \mathsf{BR}_{d_B}$. In this case $d_F$ and $d_B$ are the depths of the bounded traversals in the forward and backward directions respectively. The enlarged set $\mathsf{S}^e$ replaces $\mathsf{S}$ and the first $d_F$ time frames in the combinational unrolling. Dually for $\mathsf{T}^e$. More formally:

$$\mathsf{S}^e = \exists_{s_0..s_{d_F-1}}(\mathsf{S} \bigwedge_{i=0..d_F-1} \mathsf{TR}(s_i, s_{i+1})) \tag{5}$$

- Disproving mutual reachability for $\mathsf{S}^e$ and $\mathsf{T}^e$ is equivalent to disproving it for $\mathsf{S}$ and $\mathsf{T}$.
- Proving mutual reachability for $\mathsf{S}^e$ and $\mathsf{T}^e$, i.e., finding a counterexample connecting them, is equivalent to proving mutual reachability for $\mathsf{S}$ and $\mathsf{T}$. The generated counterexample is partial, and we need to complete it with a prefix and a suffix, in order to reach the original $\mathsf{S}$ and $\mathsf{T}$ states.

Intuitively, we may expect a performance gain from the above task partitioning if the preprocessing work done with BDDs is manageable and, more importantly, it is able to decrease the overall memory/time complexity. As BDDs are able to start a mutual reachability task from $\mathsf{S}$ and $\mathsf{T}$, the sequential depth of SAT exploration (e.g., the induction depth) can be lower with target enlargement than with the original problem. Moreover, we may expect to handle sequentially deeper problems, given the expected capacity of the SAT tool. Obviously, the above expectations strongly depend on how efficiently the enlarged state sets replace the equivalent set of time frames in the combinational unrolling. In general we may expect an advantage related to the reduction of variables and clauses in the final problem.

Let us examine the substitution on the particular case of BMC. We choose it for sake of simplicity, and we show that a given BMC problem of bound $k$ can be solved, with target enlargement, as a BMC problem with shorter bound. Similar formulations can be done with unbounded model checking. The following proposition holds.

**Proposition 4.1** *Let $BMC_k(\mathsf{TR}, \mathsf{S}, \mathsf{T})$ be a BMC problem of bound $k$, over a given transition relation $\mathsf{TR}$, an initial $\mathsf{S}$ and a target $\mathsf{T}$ state sets. Let $\mathsf{S}^e$ and $\mathsf{T}^e$ be "enlarged" initial and target sets as defined above. Let us define $\widehat{d_F}$ ($\widehat{d_B}$) the maximum value of $l_F$ ($l_B$) such that $\mathsf{FR}_{l_F} \subseteq \mathsf{S}^e$ ($\mathsf{BR}_{l_B} \subseteq \mathsf{T}^e$). Then the original $BMC_k$ problem can be solved as a new BMC problem $BMC_h(\mathsf{TR}, \mathsf{S}^e, \mathsf{T}^e)$*

*with possibly shorter bound, i.e., such that* $h = k - (\widehat{d_F} + \widehat{d_B})$.

### *4.2 Restricting SAT Search with Enlarged Target*

We now show that target enlargement can restrict the search space of the SAT solver, with additional benefits, in terms of performance, besides simple bound reduction. More specifically, let us concentrate on a particular form of BMC, the so called $exact - assume$ variant of BMC, and on the inductive steps of unbounded SAT as proposed in [23].

Exact-assume BMC can be expressed as:

$$BMC_k(\mathsf{TR}, \mathsf{S}, \mathsf{T}) = \mathsf{S}(s_0) \wedge path(s_{[0..k]}) \wedge \bigwedge_{0 \le i < k} (\neg \mathsf{T}(s_i)) \wedge \mathsf{T}(s_k) \tag{6}$$

Inductive steps in unbounded model checking [23] can be expressed as:

$$\textsc{FwdInductiveStep}_k(\mathsf{TR}, \mathsf{S}) = loopFree(s_{[0..k]}) \wedge \mathsf{S}(s_0) \wedge \bigwedge_{0 < i \le k} (\neg \mathsf{S}(s_i)) \tag{7}$$

$$\textsc{BwdInductiveStep}_k(\mathsf{TR}, \mathsf{T}) = loopFree(s_{[0..k]}) \wedge \mathsf{T}(s_k) \wedge \bigwedge_{0 \le i < k} (\neg \mathsf{T}(s_i)) \tag{8}$$

In all the above formulas, the complement of $\mathsf{S}$ and/or $\mathsf{T}$ are used to constrain the state at the $i$-th time frame. The effect of our approach is not only to constrain the SAT search space, but also to tighten the termination conditions of unbounded model checking. Our argument here is that larger start and target state sets are able to provide tighter constraints, enhancing SAT performance and enforcing termination of unbounded checks.

In the BMC case with bound $k$, all time frames but the last one are constrained by $\neg \mathsf{T}^e$. In the induction case with bound $k$, all time frames but the first one are constrained by $\neg \mathsf{S}^e$, whereas all time frames but the last one are constrained by $\neg \mathsf{T}^e$. This is clearly a stronger search space pruning than in the original BMC problem. Intuitively all states in *all* BDD-computed enlarged state sets are forbidden in *all* state paths considered by the SAT solver. In another way, a state is not considered within a SAT state path if it belongs to any BDD-based state path prefix or suffix, which is not necessarily a possible prefix or suffix of the current path in the SAT search. This search state pruning is not achievable by SAT reasoning on the original $BMC_k(\mathsf{TR}, \mathsf{S}, \mathsf{T})$ problem, due to the "linear time" reasoning employed. Moreover, the termination of unbounded induction-based verification is enforced by the smaller number of states available for loop-free state paths starting from $\mathsf{S}$ (leading to $\mathsf{T}$). The above fact directly stems from the fact that every loop free path satisfying the $\textsc{FwdInductiveStep}_k$ obtained by using the enlarged state sets is also included or equal to a loop free path satisfying the same problem generated by using the original state sets, whereas the reverse is not true.

As a final remark, notice that the complement of our target state set $\mathsf{T}$ (i.e., $\neg \mathsf{T}^e$) can be used as constraint for the $i$-th time frame (with $i < k$) in $\textsc{FwdInductiveStep}_k$ ($\mathsf{TR}$, $\mathsf{S}^e$). Moreover, $\mathsf{S}^e$ can by used dually in Bwd-

InductiveStep$_k$ ($\mathsf{TR}$, $\mathsf{T}^e$). This is not far from using an over-approximation of forward (backward) reachable states as constraint for the backward (forward) induction, as described in [15]. Nevertheless, here $\neg\mathsf{T}^e$ ($\neg\mathsf{S}^e$) is not an over-approximation of the forward (backward) reachable state set. On the contrary, it is an over-approximation of the search area where we look for states in loop-free paths. Similar considerations also hold for the BMC case.

## 5 Under-approximate Reachability and Dynamic Abstraction

Over-approximate reachability has been proposed in several works as an abstraction technique, with the aim of improving capacity and scalability. On the contrary, under-approximate techniques have received less attention in formal verification in recent years. Under-approximation was specifically addressed in the BDD sub-setting work by Somenzi et al. [21]. Many works then followed the partitioning and guided search paradigms [8,22,12] where a difficult reachability task could be faced by case splitting or focusing on a search sub-space. Within this framework, BDD sub-setting was just one of the possible ways to produce a cut on a complex state set.

The techniques we are proposing here essentially aim at using under-approximation to gather "enlarged" state sets $\mathsf{S}^e$and $\mathsf{T}^e$, such that:

- They are included in the exact forward and backward reachable state sets:

$$\begin{aligned} \mathsf{S} \subseteq \mathsf{S}^e \subseteq \mathsf{FR}(\mathsf{S}) \\ \mathsf{T} \subseteq \mathsf{T}^e \subseteq \mathsf{BR}(\mathsf{T}) \end{aligned} \tag{9}$$

- Their characteristic functions can be computed and represented in terms of BDDs at a manageable cost.

Among the various available choices, we present here:

- *Bounded traversals.*
- A new form of high-density BDD sub-setting that we call *high-density dynamic abstraction.*

The former strategy is a very straightforward option, particularly suited for symbolic traversals characterized by affordable initial iterations. With the latter one we tackle BDD explosion more aggressively. As BDD blow up is often related to the number of support variables, i.e., the variables BDDs depend on, we aim at reducing the support of state sets, with a possibly minimal impact on the number of represented state sets.

### *5.1 Under-Approximation by Bounded Traversal*

We call bounded traversal a simple variant of a standard forward and/or backward traversal, where BDD-based breadth-first reachability stops before

reaching a fix-point. Threshold based BDD control is a natural exit condition, but other options are possible as well, like cardinality of the reached state set, number of support variables, and pre-determined number of traversal iterations.

In all cases, the traversal ends up computing $\mathsf{FR}_{d_F}$ and $\mathsf{BR}_{d_B}$, and the traversal depths $d_F$ and $d_B$ are the exact parameter required to decrease the length of SAT checks.

In general, forward reachable state sets depend on all variables since it is the first iteration, whereas backward state sets follow the so called Cone-Of-Influence of the property. In practice the number of support variables (and the BDD size) of $\mathsf{T}^e$ grows for growing values $d_B$. The good choice is a trade-off between BDD size and number of state variable we are able to further constrain in successive SAT processing.

### *5.2 Under-Approximation by High-Density Dynamic Abstraction*

Our target in this section, is to adopt some optimization to gather as more states as possible in the enlarged sets before switching to SAT. As opposed to partitioning strategies, where a complex task is split in subtasks, sub-setting here means that, given a large BDD, we select a "relevant" subset, that we use for further processing steps. The pruned subset is either temporarily or permanently removed from the traversal process, since it is deemed completely or almost completely irrelevant.

We work within the inner loop of a BDD-based traversal. Whenever a state set violates a predefined threshold (BDD size and/or support size), we dynamically operate sub-setting. High density as introduced in [21] aimed at clipping a BDD so that a minimal number of minterms (i.e., states) was removed from it. Density was defined as the ratio number between the minterms in the subset and in the original BDD. Pruning was done recursively, so no particular care was taken at reducing the amount of variables in the support.

The sub-setting technique we propose is a compromise between support reduction and high-density. It can be used either for BDD super-setting or sub-setting, as the basic step is variable quantification. If *sub-setting* is our goal, we *universally quantify* a variable so that a minimal number of minterms are removed from the original BDD. Super-setting would be achieved in a dual way, by adopting existential quantification and minimizing the newly introduced minterms.

Let us suppose a function $f(X)$ is represented as a BDD, and let us use the notation $|f|$ to indicate its minterm count (i.e., the number of domain points where $f = 1$). We compute the subset:

$$f_\sigma^-(X - \sigma) = \forall_\sigma f(X) = f(x)_{\sigma=0} \wedge f(x)_{\sigma=1} \tag{10}$$

The $\sigma \in X$ variable is selected so that $|f^-|/|f|$ is maximal. We loop through such sub-setting steps until we get a sufficient reduction either in terms of BDD

size or support variables. The variable selection criteria is clearly greedy, as a sequence of optimal variable selection is not guaranteed to produce the best overall result. We should also notice that, although variable quantification does not guarantee BDD size reduction, it often does so. To take into account possible BDD size increase, our variable selection is a weighted compromise between size and density.

We call our sub-setting *dynamic abstraction* as opposed to over-approximate reachability and abstraction-refinement techniques, where variable abstraction schemes are generally decided *statically* as a pre-processing and/or post-processing step of entire traversals. Here we do variable selection and universal abstraction "on-the-fly", within inner steps of a traversal procedure. The main motivation for selecting this kind of sub-setting is that we control BDD explosion (as in other abstraction schemes) by reducing the number of support variables. We still resort to a high density heuristic, as we want to maximize the amount of state space "covered" by the enlarged set.

Due to the above mentioned dynamic scheme, time and memory efficiency of variable selection is a key issue, as the introduced overhead should by negligible within the overall traversal process. A naive approach consists of first computing variable abstraction for all variables, then selecting the one with best weighted size-density benefit. This can be very expensive, as it means to compute a new (possibly larger) BDD for each variable.

An alternative and much more effective approach is density estimation without computing the abstraction. Given $f(X)$ and a candidate variable $x_i \in X$, we can compute $|f^-|$ through a variant of the standard minterm count routine, that visits the subset of BDD nodes in $f(x)$ reachable both under the $x_i = 1$ and $x_i = 0$ constrains. A double "linear" visit can achieve this task. In the first visit we mark the $f(x)$ nodes reachable with $x_i = 0$. The second visit achieves the actual minterm count, by working on the previously marked nodes, under the opposite constraint $x_i = 1$.

As a result, a best density abstraction variable can be computed in linear time without generating any new BDD. This does not take into account the BDD size of the result. So we add an extra step where we actually compute the abstraction for the topmost variables, after ordering them by estimated minterm density. Possible BDD blow up is avoided by a size threshold controlling universal abstraction as a try-and-abort operator.

## 6 Overall Verification Algorithms

The techniques described in the previous sections are integrated within a mixed BDD/SAT verification framework using a mix of bounded (Section 5.1) and high-density dynamic abstraction (Section 5.2) traversal. The pseudo-code in Figure 2 shows the unbounded verification procedure based on induction. BMC is easily derived as a simplified version of the shown pseudo-code, where inductive checks are removed, and the main SAT verification loop stops

```
InductiveMcWithTE (TR, S, T)
    Set under-approx thresholds and bounds
    S^e= FwdUnderApproxTrav (TR, S)
    T^e= BwdUnderApproxTrav (TR, T)
    h = 0
    while (true)
        (result, cex) = BMC_h(TR, S^e, T^e)
        if (result = Sat)
            prefix = ComputeCex (TR, S, S^e, cex)
            suffix = ComputeCex (TR, T, T^e, cex)
            return (prefix, cex, suffix)
        result = FwdInductiveStep_{h+1} (TR, S^e)
        if (result = UnSat)
            return (Pass)
        result = BwdInductiveStep_{h+1} (TR, T^e)
        if (result = unsat)
            return (Pass)
        h = h+1
```

Fig. 2. Inductive verification with target enlargement.

at a predefined bound.

Initial BDD-based traversals compute enlarged sets $\mathsf{S}^e$ and $\mathsf{T}^e$. SAT verification iterates over BMC and inductive steps until either a counterexample is found by BMC or an inductive step is unsat. If a counterexample is found, it is extended by a prefix and a suffix, computed within $\mathsf{S}^e$and $\mathsf{T}^e$. If verification passes, the procedure returns the depth of termination.

Figure 2 basically shows that BDD traversals are done as a pre-processing step outside the loop over increasing bound $h$. This means that the overhead due to BDD manipulation decreases as far as verification requires longer and longer depths.

The procedure hides some details, as BDD to CNF conversion, required for generate the BMC and inductive problems with the enlarged initial and target sets. In principle, any BDD can be converted to CNF with an amount of variables and clauses linear in the BDD size. We adopt a more sophisticated approach, as described in [9], with heuristics trading off the number of clauses and of new variables required by the BDD to CNF conversion.

## 7 Experimental Results

This section describes an experimental comparison in terms of BMC and inductive verification with and without the technique described in this paper. The presented methodology is implemented within the industrial Intel tool *BOolean VErifier* (BOVE) [4]. We do not compare our results with any other tool because we want to discuss only the improvement obtained exploiting the described method over the standard methodology. Furthermore, due to the specific features implemented in BOVE (e.g., ternary encodings, gated clocks,

initial state set evaluated during an initial synchronization sequence computation [20], etc.), any comparison would be unfair.

| Model | # SV | Property | B | BOVE Original | | | | BOVE with Target Enlargement | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | # Var. | # Clauses | Mem. [MByte] | Time SAT | BBwd | Mem. [MByte] | Time [sec] Trav. | Time [sec] SAT |
| am2901 | 68 | $P_1$ Pass | 10 | 23252 | 67413 | – | *ovf* | 2 | 94 | 223 | 1653 |
| | | $P_1$ Fail | 16 | 37112 | 107709 | – | *ovf* | 2 | – | 223 | *ovf* |
| philo$_{60}$ | 120 | $P_1$ Pass | 20 | 94139 | 276838 | 72 | 745 | 2 | 74 | 151 | 591 |
| | | $P_1$ Pass | 40 | 187859 | 552778 | 173 | 2561 | 2 | 129 | 151 | 1264 |
| FIFOs | 142 | $P_1$ Pass | 14 | 31227 | 89028 | 80 | 1395 | 2 | 82 | 231 | 494 |
| | | $P_1$ Pass | 16 | 35551 | 101396 | – | *ovf* | 2 | 163 | 231 | 1659 |
| s15850.1 | 534 | $P_1$ Pass | 75 | 351424 | 969151 | 96 | 142 | 5 | 81 | 5 | 28 |
| | | $P_1$ Fail | 76 | 356097 | 982054 | 97 | 130 | 5 | 85 | 5 | 48 |
| s13207.1 | 638 | $P_1$ Pass | 109 | 344363 | 919464 | 144 | 580 | 4 | 89 | 3 | 157 |
| | | $P_1$ Fail | 110 | 347514 | 927887 | 121 | 384 | 4 | 108 | 3 | 283 |
| | | $P_2$ Pass) | 215 | 678369 | 1812302 | – | *ovf* | 10 | 216 | 5 | 1182 |
| | | $P_2$ Fail | 216 | 681520 | 1820725 | – | *ovf* | 10 | – | 5 | *ovf* |

Table 1
SAT-based Bounded Model Checking: Comparison between standard and target-enlarged BMC. *ovf* means overflow on memory or time (memory limit 512 MBytes, time limit 3600 sec). – means data not available.

We present experiments on two separate sets of circuits:

- Standard benchmarks: ISCAS'89 [6], ISCAS'89-addendum [1], and circuits belonging to the VIS distribution [5].
- Industrial designs: The IBM Formal Verification Benchmark Library [18]. This suite includes 75 circuits in Blif format. They contain from 95 to 917 memory elements. For each of them, there is a unique output (formula_1) which also indicates the property to check.

Among the previous sets we selected all verification instances hard-enough to be improved with our technique. All our data are collected on a Pentium IV 1700 MHz Workstation equipped with 1 GByte main memory, running RedHat Linux 7.1. We use a time limit of 1 hour for all experiments.

Tables 1 and 2 report our results on BMC and inductive verification respectively.

In both the tables the models are sorted by number of state variables (column # SV). For each design we present a set of properties denoted by $P_1$, $P_2$, etc. For each property we indicate the maximum sequential depth explored, i.e., the length of the counterexample B (bound). Within the reported bound, the properties are labeled according to the result they produce: They are either proved correct and denoted by Pass, or they are falsified and labeled by Fail [3].

The subsequent columns report statistics for the original and proposed methods, i.e., number of variables and clauses, memory and time used. More

[3] For Table 1, where we consider only BMC, when Pass, the property is, of course, proved correct only up to the given bound.

| Model | # SV | Property | BOVE Original | | | | | BOVE with TE | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | B | Time | | | | B | Time | | | |
| | | | | BMC | IBwd | IFwd | Total | | BMC | IBwd | IFwd | Total |
| 29_batch | 95 | $P_1$ Pass | 32 | 1 % | 83 % | 16 % | $ovf$ | 40 | 5 % | 77 % | 18 % | $ovf$ |
| 18_batch | 143 | $P_1$ Pass | 39 | 11 % | 71 % | 18 % | 3151 | 35 | 13 % | 68 % | 19 % | 1826 |
| 16_1_batch | 150 | $P_1$ Pass | 31 | 16 % | 49 % | 35 % | $ovf$ | 34 | 15 % | 55 % | 30 % | $ovf$ |
| 02_1_batch_3 | 161 | $P_1$ Pass | 28 | 12 % | 42 % | 46 % | $ovf$ | 37 | 19 % | 40 % | 41 % | 3354 |
| 19_batch | 181 | $P_1$ Pass | 19 | 23 % | 57 % | 20 % | 2786 | 16 | 20 % | 56 % | 24 % | 1022 |
| 22_batch | 191 | $P_1$ Fail | 15 | 4 % | 83 % | 13 % | 536 | 15 | 12 % | 72 % | 16 % | 197 |
| 04_batch | 256 | $P_1$ Pass | 25 | 2 % | 80 % | 18 % | $ovf$ | 28 | 8 % | 74 % | 18 % | $ovf$ |

Table 2
Induction-based Verification: Comparison between standard and target-enlarged induction. $ovf$ means overflow on memory or time (memory limit 512 MBytes, time limit 3600 sec).

specifically, BOVE Original indicates our implementation (within the BOVE environment) of standard BMC [3] and inductive [23] verification. BOVE with TE is the previous version improved with the target enlargement methodology described in this paper.

Within Table 1 we used the bounded traversal technique in order to get the enlarged sets, so we report the number of backward reachability steps performed (column BBwd), and the Trav. Time needed to conclude this phase. Notice that in this table we compare experiments using BOVE Original and BOVE with TE with the same value of the bound B. This table shows a maximum gain of about 5X for the first experiment, and a good improvement for most of the benchmarks.

For Table 2 we used the dynamic abstraction technique, so that column B represents the depth reached by BOVE when the result was found or an overflow occurred for the two methodologies respectively. In particular, for enlarged case, the meaning of this column must be intended as the sum of the maximum time step analyzed with SAT and those provided by BDDs ($k = h + d_F + d_B$). Columns BMC, IBwd, and IFwd indicate the percentage of time dedicated to the BMC, backward, and forward induction steps respectively over the total amount of Total time. Statistics about the BDD manipulations are not reported.

Analyzing these data, the first observation we do is that the backward inductive step is almost always the hardest check. Indeed, in our experiments, this step was always the one providing the proof of correctness (when the property was true). Nevertheless, with our new technique, the weights of BMC and the inductive (backward) check have been slightly balanced. At first glance, it could seem that BMC now is more costly than with the original method, but, in absolute terms, the time required for each check has been usually reduced. Notice also that we could almost always explore a deeper time step.

Results are still preliminary yet encouraging for both bounded and unbound SAT-based verification. While in some cases we drastically reduced

the verification time (up to a 5x factor), on others we could substantially increment the bound (up to almost 30%) analyzed in the same amount of time, i.e., before expiring system resources.

## 8 Conclusions and Future Works

BDDs and SAT-solvers are the most widely used core technologies within the verification domain. In this paper we propose an idea to exploit the best of this two methodologies. First of all, we perform symbolic forward and backward reachability analysis to partially carry out the verification task. This step also enlarge the initial and the target sets of states. These enlarged sets are subsequently used within a SAT-based bounded model checking or inductive verification phase. Preliminary experimental results show the potentiality of the approach within academic and industrial tools.

Among the possible future work we envisage two main tasks. First of all, we have to check the methodology on a broader set of experiments possibly using real verification instances from the industrial Intel designs. Secondly, we envisage the possibility to integrate the method with an abstraction–and–refinement approach.

## References

[1] *MCNC Private Communication.*

[2] Baumgartner, J. and A. Kuehlmann, *Enhanced Diameter Bounding Via Structural Transformation*, in: *Proc. Design Automation & Test in Europe Conf.*, Paris, France, 2004.

[3] Biere, A., A. Cimatti, E. M. Clarke, M. Fujita and Y. Zhu, *Symbolic Model Checking using SAT procedures instead of BDDs*, in: *Proc. 36th Design Automat. Conf.*, New Orleans, Louisiana, 1999, pp. 317–320.

[4] Bischoff, G. P., K. S. Brace, S. Jain and R. Razdan, *Formal Implementation Verification of the Bus Interface Unit for the Alpha 21264 Microprocessor*, in: *Proc. Int'l Conf. on Computer Design*, Austin, Texas, 1997.

[5] Brayton, R. K., et al., *VIS*, in: M. Srivas and A. Camilleri, editors, *Proc. Formal Methods in Computer-Aided Design*, LNCS **1166** (1996), pp. 248–256.

[6] Brglez, F., D. Bryan and K. Koźmiński, *Combinatorial Profiles of Sequential Benchmark Circuits*, in: *Proc. IEEE ISCAS'89*, 1989, pp. 1929–1934.

[7] Bryant, R. E., *Graph–Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on Computers **C–35** (1986), pp. 677–691.

[8] Cabodi, G., P. Camurati, L. Lavagno and S. Quer, *Disjunctive Partitioning and Partial Iterative Squaring: an effective approach for symbolic traversal of large*

*circuits*, in: *Proc. 34th Design Automat. Conf.*, Anaheim, California, 1997, pp. 728–733.

[9] Cabodi, G., S. Nocco and S. Quer, *Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals*, in: *Proc. Design Automation & Test in Europe Conf.*, Munich, Germany, 2003, pp. 898–903.

[10] Cho, H., G. D. Hatchel, E. Macii, B. Plessier and F. Somenzi, *Algorithms for Approximate FSM Traversal Based on State Space Decomposition*, IEEE Trans. on Computer-Aided Design **15** (1996), pp. 1465–1478.

[11] 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, *Proc. Computer Aided Verification*, LNCS **2102** (2001), pp. 435–453.

[12] Ganai, M. K., A. Aziz and A. Kuehlmann, *Enhancing Simulation with BDDs and ATPG*, in: *Proc. 36th Design Automat. Conf.*, New Orleans, LA, 1999, pp. 385–390.

[13] Govindaraju, S. G., D. L. Dill, A. Hu and M. A. Horowitz, *Approximate Reachability Analysis with BDDs using Overlapping Projections*, in: *Proc. 35th Design Automat. Conf.*, San Francisco, California, 1998, pp. 451–456.

[14] Gupta, A., M. Ganai, C. Wang, A. Yang and P. Ashar, *Learning from BDDs in SAT-based Bounded Model Checking*, in: *Proc. 40th Design Automat. Conf.*, Anaheim, CA, 2003, pp. 824–829.

[15] Gupta, A., M. Ganai, C. Wang, Z. Yang and P. Ashar, *Abstraction and BDDs Complement SAT–Based BMC in* Diver, in: W. A. H. Jr. and F. Somenzi, editors, *Proc. Computer Aided Verification*, LNCS **2725** (2003), pp. 206–209.

[16] Gupta, A., Z. Yang, P. Ashar and A. Gupta, *SAT–Based Image Computation with Application in Reachability Analysis*, in: *Proc. Formal Methods in Computer-Aided Design*, LNCS **1954**, Austin, TX, USA, 2000.

[17] Gupta, A., Z. Yang, P. Ashar, L. Zhang and S. Malik, *Partition–Based Decision Heuristic for Image Computation using SAT and BDDs*, in: *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, 2001.

[18] IBM Formal Verification Benchmark Library, *http://www.haifa.il.ibm.com/projects/verification/rb_homepage/fvbenchmarks.html*.

[19] McMillan, K. L., *Interpolation and SAT-Based Model Checking*, in: W. A. H. Jr. and F. Somenzi, editors, *Proc. Computer Aided Verification*, LNCS **2725**, Boulder, CO, USA, 2003, pp. 1–13.

[20] Pixley, C., S. W. Jeong and G. D. Hachtel, *Exact Calculation of Synchronization Sequences Based on Binary Decision Diagrams*, in: *Proc. 29th Design Automat. Conf.*, 1992, pp. 614–619.

[21] Ravi, K. and F. Somenzi, *High–Density Reachability Analysis*, in: *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, 1995, pp. 154–158.

[22] Ravi, K. and F. Somenzi, *Hints to Accelerate Symbolic Traversal*, in: *Correct Hardware Design and Verification Methods (CHARME'99)*, LNCS **1703** (1999), pp. 250–264.

[23] Sheeran, M., S. Singh and G. Stålmarck, *Checking Safety Properties Using Induction and SAT Solver*, in: W. A. Hunt and S. D. Johnson, editors, *Proc. Formal Methods in Computer-Aided Design*, LNCS **1954** (2000), pp. 108–125.

[24] Yang, C. H. and D. L. Dill, *Validation with Guided Search of State Space*, in: *Proc. 35th Design Automat. Conf.*, San Francisco, California, 1998.

[25] Zhang, L. and S. Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, in: E. Brinksma and K. G. Larsen, editors, *Proc. Computer Aided Verification*, LNCS **2404**, Cophenagen, Denmark, 2002, pp. 17–36.

# An Incremental Algorithm to Check Satisfiability for Bounded Model Checking [1]

HoonSang Jin [2] Fabio Somenzi [3]

*University of Colorado at Boulder*

**Abstract**

In Bounded Model Checking (BMC), the search for counterexamples of increasing lengths is translated into a sequence of satisfiability (SAT) checks. It is natural to try to exploit the similarity of these SAT instances by *forwarding* clauses learned during conflict analysis from one instance to the next. The methods proposed to identify clauses that remain valid fall into two categories: Those that are oblivious to the mechanism that generates the sequence of SAT instances and those that rely on it. In the case of a BMC run, it was observed by Strichman [20] that those clauses learned during one SAT check that only depend on the structure of the model remain valid when checking for longer counterexamples. Eén and Sörensson [9] pointed out that all learned clauses can be forwarded if the translation into SAT obeys commonly followed rules. Many clauses that are forwarded this way, however, are of little usefulness and may degrade performance. This paper presents an extension to Strichman's approach in the form of a more general criterion to filter conflict clauses that can be profitably forwarded to successive instances. This criterion, in particular, is still *syntactic* and quite efficient, but accounts for the presence of both *primary* and *auxiliary objectives* in the SAT instance. This paper also introduces a technique to *distill* clauses to be forwarded even though they fail the syntactic check. Distillation is a *semantic* approach that can be applied in general to incremental SAT, and often produces clauses that are independent of the primary objective, and hence remain valid for the remainder of the sequence of instances. In addition, distillation often improves the quality of the clauses, that is, their ability to prevent the examination of large regions of the search space. Experimental results obtained with the CirCUs SAT solver confirm the efficacy of the proposed techniques, especially for large, hard problems.

*Key words:* bounded model checking, propositional satisfiability, conflict-learned clauses, incremental algorithms.

[1] This work was supported in part by SRC contract 2003-TJ-920.

[2] Email: Jinh@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] determines whether for model $\mathcal{K}$ there exists a counterexample to property $\varphi$ of length less than or equal to $k$. If such a counterexample is found, or if $k$ is large enough [19,18,6,1], then BMC effectively answers the question $\mathcal{K} \models \varphi$; otherwise, it increases the user's confidence in the correctness of $\mathcal{K}$.

BMC converts the search for a counterexample of a certain maximum length into a sequence of propositional satisfiability (SAT) checks. In its simplest form, the length starts from 0 and is incremented by 1 for each instance. At the $i$-th step of this iteration, a propositional formula is built from $\mathcal{K}$ and $\varphi$ that is satisfiable if and only if there exists a counterexample to $\varphi$ in $\mathcal{K}$ of length $k = i - 1$. Though variations on this scheme are easy to envisage, and can be accommodated by the techniques discussed in this paper, we shall limit our discussion to this case. Each formula checked for satisfiability consists of three parts, corresponding to the initial state constraint, the unrolled transition relation, and the property to be satisfied. In the last part, one often distinguishes a *primary objective* (e.g., the last state of the counterexample violates an invariant) from *auxiliary objectives* (e.g., no state except the last one violates that invariant). Auxiliary objectives express information about the problem gathered from failed attempts to find shorter counterexamples. They may help in directing the search process.

The emergence of efficient SAT solvers over the last decade [21,25,26,11] has greatly contributed to the success of BMC. The new generation of SAT solvers improves over the classical DPLL procedure [8,7] in several ways. Of interest to us are conflict analysis and clause recording: When a conflicting assignment is found, it is analyzed to identify a subset that is still conflicting. The disjunction of the negation of the literals in the subset is a *conflict-learned clause* (or, more concisely, a *conflict clause*) that can be added to the given SAT instance to prevent the examination of regions of the search space that are guaranteed to contain no solutions. Not all conflict clauses are worth keeping; many SAT solvers periodically discard those that have proved ineffective.

Incremental SAT solvers [24,20,9] try to leverage the similarity between the elements of a sequence of SAT instances; most do so by re-utilizing conflict clauses, though when many closely related instances must be solved, caching solutions [15] and incremental translation [2] can also be effective. If a SAT instance is obtained from another by adding some clauses (as in [12]), then all conflict clauses of the first can be *forwarded* to the second. This is correct because the second instance implies the first, which in turn implies all its (conflict) clauses. Therefore, when clauses are only added through the sequence of instances, there is no need to screen conflict clauses to determine which ones can be forwarded. This, on the other hand, is necessary when arbitrary clauses are both added and subtracted to create a new instance. A common approach to such general case is to have the incremental SAT solver

keep track of whether a conflict clause depends on some removed clauses. The approach of [24] is to record, for each conflict clause, the clauses that made up the corresponding implication graph. This approach does not require any foreknowledge of the subsequent SAT instances to be solved incrementally, and does not restrict the changes possible from one instance to the next; however, keeping track of dependencies may be expensive.

Strichman [20] was the first to observe that in BMC some clauses are known to survive through all instances in the sequence. A formula passed by BMC to the SAT solver contains clauses that describe the transition relation of the model unrolled a number of times. These clauses are not discarded when the length of the counterexample is increased. Hence, a conflict clause that depends only on them can be forwarded. The advantage of this approach is that complete dependence information is no longer needed; one-bit marker per clause is sufficient. Such a marker is derived from the structure of the implication graph that produces the clause. Therefore, we speak of a *syntactic* criterion in this case.

The authors of [9] remarked that tracking dependencies is not required if only *unit* clauses are removed. Such clauses can be regarded as assumptions by the SAT solver. As a result, a conflict clause incorporates its assumptions or some of their implied literals, and is not invalidated when the assumptions are repealed. It was further observed in [9] that the standard encoding of objectives into SAT guarantees that the clauses that must be removed when the counterexample length is incremented are unit clauses. Hence, all conflict clauses can be forwarded. The approach of [9] exemplifies those incremental satisfiability algorithms that are aware of the mechanism generating the sequence of SAT instances. On the other hand, when one of its unit clause assumptions is reversed, a conflict clause becomes satisfied and therefore inert.

Having many inert clauses in the solver may significantly affect performance. Therefore we want to forward only clauses that have a good chance of remaining active in successive instances. To this purpose, we propose a *syntactic* criterion that improves on the one of [20] in two ways. First, it accounts for *auxiliary objectives*, and hence can forward more clauses. Second, it does not require the examination of the entire implication graph when marking a conflict clause.

We also present a *semantic* forwarding criterion, which *distills* the clauses that cannot be forwarded according to the syntactic check into clauses implied by the new instance. These distilled clauses are sometimes independent of the objective of the new instance and usually more effective than the raw clauses from which they are derived in preventing exploration of fruitless regions of the search space.

The rest of this paper is organized as follows. Section 2 reviews background material. Section 3 describes the incremental SAT algorithm, while Section 4 discusses the experiments conducted to assess its effectiveness. Section 5 concludes.

## 2 Preliminaries

Let $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$ be sets of Boolean variables. We designate by $V'$ the set $\{v'_1, \ldots, v'_n\}$ consisting of the primed version of the elements of $V$, and by $V^i$ the set $\{v^i_1, \ldots, v^i_n\}$. Likewise, $W^i = \{w^i_1, \ldots, w^i_m\}$. An *open system* is a 4-tuple

$$\Omega = \langle V, W, I, T \rangle \ ,$$

where $V$ is the set of (current) state variables, $W$ is the set of combinational variables, $I(V)$ is the initial state predicate, and $T(V, W, V')$ is the transition relation. The variables in $V'$ are the next state variables. All sets are finite, and all variables range over finite domains.

Bounded Model Checking (BMC) [3] reduces the search for a counterexample to a linear time property to propositional satisfiability. Given an open system $\Omega$, an LTL [17] formula $\varphi$, and a bound $k$, BMC tries to refute $\Omega \models \varphi$ by proving the existence of a witness of length $k$ to the negation of the LTL formula.

BMC generates a propositional formula $[\![\Omega, \neg\varphi]\!]_k$ that is satisfiable if and only if a counterexample to $\varphi$ of length $k$ exists in $\Omega$; $[\![\Omega, \neg\varphi]\!]_k$ is defined as follows:

$$[\![\Omega, \neg\varphi]\!]_k = I(V^0) \wedge \bigwedge_{0 \le i < k} T(V^i, W^i, V^{i+1}) \wedge [\![\neg\varphi]\!]_k \ , \tag{1}$$

where $[\![\neg\varphi]\!]_k$ expresses the satisfaction of $\neg\varphi$ along that path. (See [3] for the details of the translation.) It is customary to write $[\![\neg\varphi]\!]_k$ as $\omega_k \wedge F_k$, where $\omega_k$ is a literal called the *primary objective*. If it is known that $[\![\Omega, \neg\varphi]\!]_j$ is unsatisfiable for $j < k$, then one can conjoin (1) with

$$\bigwedge_{0 \le i < k} \neg[\![\neg\varphi]\!]_i \ . \tag{2}$$

Each term $\neg[\![\neg\varphi]\!]_i$ is written $\neg\omega_i \wedge F_i$, where $\neg\omega_i$ is an *auxiliary objective*.

A SAT solver returns assignments to the variables of a propositional formula that satisfy it, if such assignments exist. A *literal* is either a variable or its complement, a *clause* is a disjunction of literals from distinct variables, and a *conjunctive normal form* (CNF) formula is a conjunction of clauses. An *And-Inverter Graph* (AIG) [16] is a Boolean circuit such that each node's output is the conjunction of its two inputs. The arcs of the circuit may be *complementing*. A Binary Decision Diagram (BDD) [5] is a Boolean circuit such that each node is a multiplexer controlled by an input variable. Most SAT solvers operate on a propositional formula in CNF. Our SAT solver CirCUs [14,13], on the other hand, accepts a combination of CNF clauses, AIG, and reduced, ordered BDDs. Each result of a conflict analysis is represented as one clause [10]. Hence, the algorithms described in this paper can be applied to any SAT solver based on clause recording.

```
1  DPLL() {
2      while (ChooseNextAssignment())
3          while (Deduce() == CONFLICT) {
4              blevel = AnalyzeConflict();
5              if (blevel ≤ 0) return UNSATISFIABLE;
6              else Backtrack(blevel);
7          }
8      return SATISFIABLE;
9  }
```

Fig. 1. DPLL algorithm with conflict analysis

Figure 1 shows the pseudocode of the DPLL procedure as implemented in most modern SAT solvers, including CirCUs. The algorithm maintains a list of assignments that is initialized with the unit clauses from the SAT instance. If all variables have been given a value, a satisfying set of assignments has been found. Otherwise, an assignment is either extracted from the list, or created by a new decision; it is added to the assignment stack, and its consequences are deduced. Every time a new decision is made, the *decision level*, which is initially 0, is incremented. If a conflict is detected, it is analyzed. The results of the analysis are a conflict clause and a backtracking decision level. The latter tells how much of the assignment stack should be erased (decreasing the decision level) before continuing the search.

Conflict analysis relies on the *implication graph*, which is a directed acyclic graph (DAG) whose nodes are the variables in the current set of assignments plus a special *conflict node* if the assignments are conflicting. The arcs of the DAG are such that if the predecessors of node $\nu$ are $\nu_1, \ldots, \nu_m$, then there exists a clause, an AIG node, or a BDD, such that it implies the value of $\nu$ given the values of $\nu_1, \ldots, \nu_m$. The predecessors of the conflict node identify a clause, AIG node, or BDD, whose assignments are inconsistent. A root of the graph corresponds to a decision assignment. Note that different implication graphs may be obtained from the same set of assignments, depending on the order in which their implications are propagated. A conflict clause is obtained by disjoining the negation of the literals forming a cut in the implication graph that separates the conflict node from the roots of the graph. The First Unique Implication Point (UIP) approach [26] starts from the conflict node and looks for the first cut such that it contains exactly one literal implied by the most recent decision.

Every non-root node of the implication graph identifies an *antecedent* clause: The implied value of the node contributes one literal, and the negation of the predecessor values supplies the others. Some of these clauses correspond to clauses in the input description or were derived from previous conflicts. Others come from AIG nodes or BDDs. For instance, an AIG node $a = b \wedge c$, and assignment $a = 1$ implying $c = 1$ implicitly give the clause $(\neg a \vee c)$. The conflict clause is obtained by successive resolutions starting from the conflicting clause associated to the conflict node. At each step one literal implied at the

current decision level is resolved using its antecedent clause. All the clauses involved in the resolution are implied by the function whose satisfiability is being checked.

## 3 Forwarding Clauses

We consider an incremental SAT algorithm that exploits the similarities among SAT instances that form a sequence by using the conflict clauses generated from the previous instances to guide the search for a solution to the current instance. We assume that the second and successive instances of the sequence are obtained by removing some clauses from the instances immediately preceding them, and then adding some other clauses.

In BMC the unsatisfiability of a SAT problem usually comes from the simultaneous constraining of the initial and final state of a path because the formula representing the unrolled transition relation and the constraint on the initial states is normally satisfiable. However, this does not mean that the conflicts the solver goes through in proving unsatisfiability involve variables from most time frames. First, there may be conflicts due to inconsistent assignments to the inputs and outputs of some circuit elements. Second, the proof of unsatisfiability may rely on conflicts that establish non-trivial facts about intermediate states of possible counterexamples, given the constraints on the initial states. Figure 2 provides some intuition for how *local* conflicts arise. It shows an AIG produced by unrolling a transition relation twice. The property being checked is an invariant. The three parts of the figure are three snapshots taken during the search. Each circle is a node of the AIG. A circle is filled if the node is assigned a value. The three snapshots suggest that the implication graph initially consists of several connected components. If a conflict occurs when extending one of the components not including the objective (the diamond at the far right), then the resulting conflict clause is totally independent of the current objective and is a good candidate for forwarding.

As recalled in Section 1, it was noted in [9] that when objectives are identified by literals, all conflict clauses can be forwarded. However, a clause that contains the old primary objective, is trivially satisfied when that objective is turned into auxiliary by negating its literal. In general, the usefulness of conflict clauses that depend on the primary objective is dubious, even when they do not contain the objective literal. Hence, in the following, we propose two techniques to identify *objective-independent* clauses.

### *3.1 Objective Tracing*

We are interested in extending the criterion of [20] to account for auxiliary objectives since they contribute to many conflicts, especially when looking for looping counterexamples.

Fig. 2. Examples of justification clouds

**Definition 3.1** Let $[\![\neg\varphi]\!]_k = \omega_k \wedge F_k$. A conflict clause $\gamma$ is *objective-dependent* if $\omega_k$ is an ancestor of the conflict node in the implication graph, or at least one objective-dependent clauses is used in its resolution. Otherwise $\gamma$ is *objective-independent*.

We show an AIG and an implication graph for it in Figure 3. Each horizontal line in Figure 3(a) represents an AIG node; a dot stands for complementation. The objective is $x_9$ and a conflict happens after three decisions have been made for $x_1$, $x_4$, and $x_3$. Along with the implications, we also propagate the objective flag through the implication graph. For example, we mark $x_7$ and $x_8$ because they are implied by the objective. The dotted line in Figure 3(b) encloses the marked nodes.

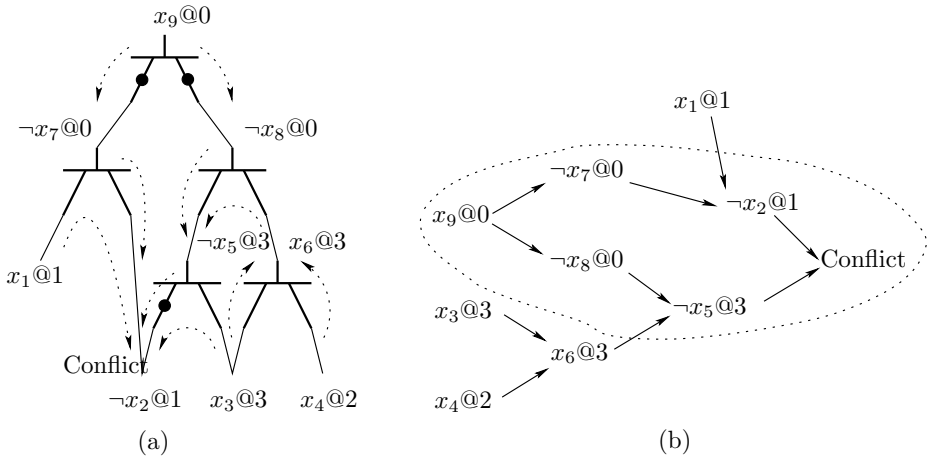


Fig. 3. Example of tracing objective

In our incremental algorithm, conflict analysis has the additional goal to check whether the conflict is related to the objective. The conflict in Figure 3 is objective-dependent, since one of the ancestors of the conflict node is $x_9$. A naive approach could identify objective-dependent conflicts by checking whether the objective is in the transitive fanin of the conflict node. However, most modern SAT solvers, including CirCUs, use the first UIP to find concise explanations of conflicts. Therefore, standard conflict analysis does not need to traverse all the transitive fanin of the conflict node. Hence the naive approach may incur overhead.

Since we propagate the objective flag during implication, we can check if the conflict is related to the objective by checking the mark of the conflict node. If the conflict node is not marked then we need to traverse the implication graph to check whether objective-dependent conflict clauses are the reason for the current conflict. However, we only traverse until the first UIP is found. Even though the rest of the implication graph has objective-dependent conflict clauses, they can be ignored. The reason of the conflict is isolated from those clauses by the first UIP.

In [20], the author propose a method to identify conflict clauses to be forwarded in BMC based solely on the circuit structure. First, all the clauses created from the circuit structure are marked. Once a conflict happens, one checks if all clauses leading to the conflict are marked. It so, the conflict is derived from inconsistency between the current assignment and the circuit structure. Therefore, the conflict is marked for forwarding. This method does not account for auxiliary objectives, which, as shown in Section 4, often speed-up BMC. Second when BMC is applied to optimized circuits, in which most redundancies have been removed, the clauses that are solely derived from the circuit structure tend to be few. This occurs in our experimental setup, since we apply BDD sweeping [16] to remove redundancy.

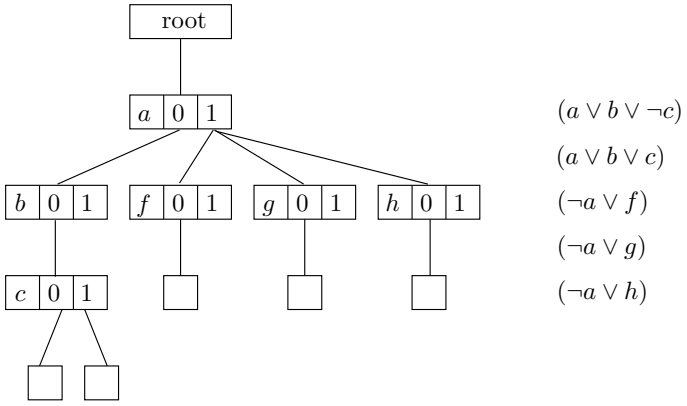


Fig. 4. Example of clause trie. Each node has two sets of children corresponding to the two literals of each variables.

### *3.2 Distillation*

Although the criterion of Section 3.1 forwards more clauses than the one of [20], it is still rather conservative and may miss many useful conflict clauses. Therefore, in this section, we develop a *semantic* criterion that is applied to small clauses that failed the syntactic check based on dependency on the objective.

To distill a clause under the new objective, we check whether the clause is satisfied under the assignments that are implied by the unit clauses of the new SAT instance. If the clause is satisfied, it is discarded. Otherwise, we assert the negations of its literals and carry out the resulting implications. If this does not result in a conflict, the clause is discarded. (Therefore, distillation can be applied also when not all clauses can be forwarded.) Otherwise, the clause obtained by conflict analysis is the distilled version of the given clause and is forwarded.

Even though we limit the number of literals in the candidate clauses, there may still be many of them. Therefore, we build a trie (cf. [25]) with the set of candidates. Figure 4 shows an example. With the trie, we can distill the clauses with at most 7 decisions, instead of the 12 decisions required if we process the clauses one by one. We do not explicitly optimize the trie when clauses can be merged. In Figure 4, $(a \vee b \vee \neg c)$ and $(a \vee b \vee c)$ could be merged into $(a \vee b)$. If the new clause indeed causes a conflict, it will be found when trying $(a \vee b \vee \neg c)$. The size of the trie depends on the order of the variables. We sort the variables according to their number of occurrences in the candidate clauses.

Figure 5 shows the distillation algorithm. For each element in the trie, the decision on the children '0' and '1' is made in DistillationAux() if there is a non-empty suffix from them. Conflict analysis is invoked when BCP() results in a conflict. If the resulting conflict clause has fewer literals than the number of trie nodes on the path from the root, it is forwarded. Otherwise the

```
1   Distillation (Trie) {
2       for each t in Trie {
3           if (Value(t.node)! = X) {
4               Distillation(t.child[Value((t.node)]);
5               continue ;
6           }
7           DistillationAux(t, 0);
8           DistillationAux(t, 1);
9       }
10  }
11
12  DistillationAux (t, value) {
13      if (t.child[value]) {
14          level = MakeDecisionBasedOnTrie(t.node, value);
15          if (BCP(level) == CONFLICT) {
16              conflictClause = ConflictAnalysis(level);
17              if (numLiterals(conflictClause) < level)
18                  AddConflictClause(conflictClause) ;
19              else
20                  AddConflictCaluseBasedOnTrieNode();
21              UndoImplication(level);
22              return ;
23          }
24          else {
25              Distillation(t.child[value]);
26          }
27      }
28  }
```

Fig. 5. Distillation algorithm

conflict clause based on the decisions that have been made is forwarded. The former case is more frequent. Since we want to go through all the trie nodes one by one, we use chronological backtracking based on the trie structure.

The distillation process has several advantages. First, it is a semantic approach that may derive clauses that were not produced by previous SAT checks. Second, some of these clauses are reusable because they do not depend on the current objective. Since the criterion based on tracing the objective is conservative, we often find many objective-independent clauses from the objective-dependent clauses of the previous run. Third, the quality of conflict clauses usually is improved by distillation. This is partly due to the different order in which assignments are made, and which results from the traversal of the trie. Moreover, the first UIP tends to be closer to the conflict than the literals in the clause to be distilled that it replaces.

A final, important advantage of distillation is that the variable scores used to make decisions are updated during the process. Therefore, distillation

biases the search based on information from the previous instances in the sequence. In [23], the entire proof of unsatisfiability from one SAT run is used to bias the variable scores of the next run. With distillation, only the part of the proof that is still useful with the new primary objective affects the scores.

## 4 Experimental Results

We have implemented the clause forwarding techniques in CirCUs, which is built on top of VIS-2.1 [4,22]. To show the effectiveness of objective tracing and distillation, we compare non-incremental SAT to incremental SAT. The non-incremental version of CirCUs was shown to be competitive with a popular CNF SAT solver, Zchaff, in [14].

The experimental setup is as follows. We build BMC instances for given LTL properties from the VIS benchmark suite [22]. We check for counterexamples of length up to 20. We first expand the AIG for the prescribed number of time frames and then apply BDD sweeping [16] to the result to remove redundancy. The maximum number of literals of a clause that undergoes distillation is 50.

All experiments have been performed on a 1.7 GHz Pentium IV with 1 GB of RAM running Linux. We have set the time-out limit to 10000 s. Two lines are drawn in each plot: the main diagonal, and the result of least square fitting for $y = a \cdot x^b$.

The left scatter plot of Figure 6 shows that using auxiliary objectives provides a rather consistent speed-up. The combined effect of all the new techniques, including auxiliary objectives, objective tracing, and distillation, is shown in the right plot of Figure 6 by comparison to non-incremental SAT.

To show the impact of distillation, we compare incremental SAT with and without distillation in the left plot of Figure 7. To justify the claim that UIP-based conflict analysis enhances the quality of conflict clauses we compare it to using the clauses as they come out of the trie. As one can see in the right plot of Figure 7, the use of UIP-based conflict analysis often generates better results.

The number of conflict clauses forwarded by several methods is shown in Table 1. The number of forwarded clauses by the method of [20] is shown in the third column. It is collected from BMC runs for all timeframes. The fourth column shows the number of forwarded clauses by the proposed objective tracing method. This method identifies many more clauses than the method of [20].

The fifth and sixth columns show the number of clauses forwarded by distillation. The objective-independent clauses from distillation are collected from all timeframes, but the objective-dependent clauses from distillation are generated from the last timeframe only. Only the objective-independent clauses are forwarded to the next runs.

To support the claim that the quality of conflict clauses is improved by

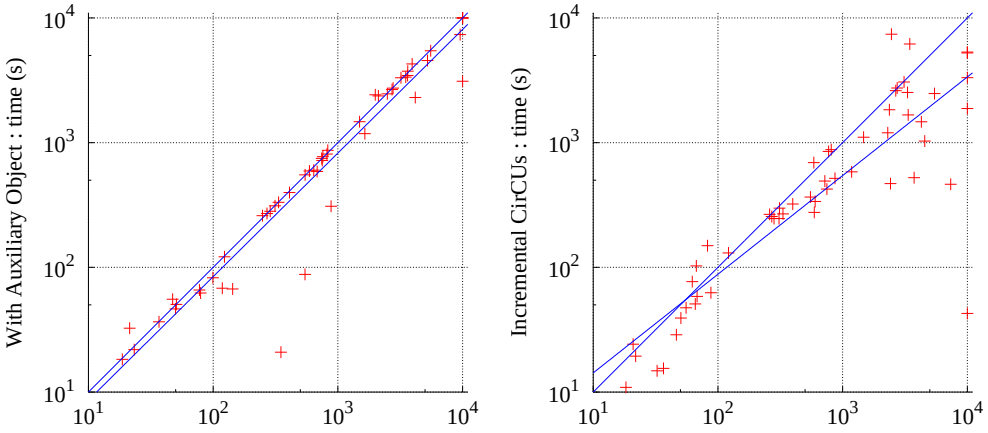


Fig. 6. Effects of auxiliary objective (left) and effects of incremental solution (right)

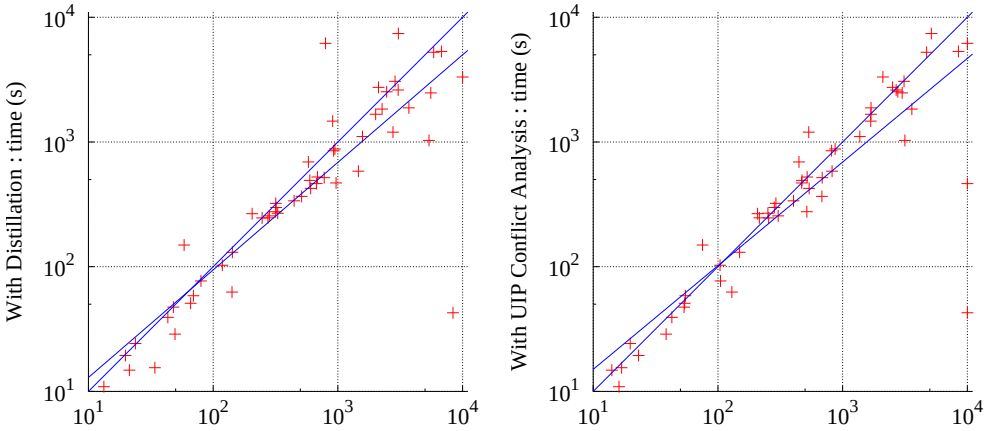


Fig. 7. Effects of distillation (left) and effects of UIP conflict analysis (right)

distillation, we show the number of literals per conflict. We achieve a reduction of approximately 10% in the number of literals per conflict.

## 5 Conclusions

We have presented two techniques for efficient incremental SAT checking in BMC. One is a syntactic technique that identifies clauses that can be profitably forwarded from one SAT instance to the next by tracing their dependence on the primary objective of the SAT problem. The other technique distills clauses that fail the tracing criterion into fewer, smaller clauses that can be

Table 1
Number of reused conflict clauses by various methods

| Design | total | [20] | Tracing | Distillation | | Lit/Conflict | |
|---|---|---|---|---|---|---|---|
| | | | | non-obj. | obj. | before | after |
| simple8 | 19350 | 76 | 353 | 2014 | 2954 | 5.7 | 5.0 |
| cups | 38292 | 720 | 1891 | 5663 | 11222 | 5.7 | 5.1 |
| blackjack | 43852 | 335 | 619 | 1883 | 2765 | 4.5 | 4.0 |
| gcd | 54383 | 437 | 1013 | 4246 | 4446 | 5.2 | 4.8 |
| two | 68255 | 422 | 979 | 2890 | 6340 | 6.0 | 5.4 |
| vending | 106898 | 361 | 1144 | 5083 | 7720 | 5.2 | 4.6 |
| goodbakery | 118323 | 268 | 636 | 4053 | 9074 | 5.3 | 4.6 |
| rcnum16 | 122263 | 591 | 1530 | 5567 | 21939 | 6.5 | 5.8 |
| spinner32 | 154788 | 298 | 3205 | 12924 | 33181 | 5.5 | 5.1 |
| all | 206398 | 248 | 1458 | 4253 | 7303 | 5.9 | 5.3 |

forwarded. Experiments indicate that the combination of these two techniques greatly increases the number of forwarded clauses over previous methods, while preventing many useless clause from cluttering the solver's data structures. This results in a significant improvement in the speed of BMC. Though we have described our techniques for a hybrid solver used for BMC, they are applicable in general to solvers based on clause recording, and to problems that benefit from an incremental approach to satisfiability.

## 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)*. Springer-Verlag, Berlin, July 2004. To appear.

[2] M. Benedetti and S. Bernardini. Incremental compilation-to-SAT procedures. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.

[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. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided*

*Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[6] E. Clarke, D. Kroening, 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] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[9] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[10] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the Design Automation Conference*, pages 747–750, New Orleans, LA, June 2002.

[11] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[12] J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1–2):177–186, Jan. 1993.

[13] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*. Springer-Verlag, Berlin, July 2004. To appear.

[14] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.

[15] J. Kim, J. Whittemore, J. P. M. Silva, and K. A. Sakallah. On solving stack-based incremental satisfiability problems. In *Proceedings of the International Conference on Computer Design*, pages 379–382, Sept. 2000.

[16] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference*, pages 232–237, Las Vegas, NV, June 2001.

[17] 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.

[18] 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.

[19] 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.

[20] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *Correct Hardware Design and Verification Methods (CHARME 2001)*, pages 58–70, Livingston, Scotland, Sept. 2001. Springer. LNCS 2144.

[21] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

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

[23] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proceedings of the Design Automation Conference*, pages 535–538, San Diego, CA, June 2004.

[24] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, Las Vegas, NV, June 2001.

[25] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.

# Making the Most of BMC Counterexamples

Alex Groce [1,2] Daniel Kroening [1,3]

*Computer Science Department*
*Carnegie Mellon University*
*Pittsburgh, PA, USA*

**Abstract**

The value of model checking counterexamples for debugging programs (and specifications) is widely recognized. Unfortunately, bounded model checkers often produce counterexamples that are difficult to understand due to the values chosen by a SAT solver. This paper presents two approaches to making better use of BMC counterexamples. The first contribution is a new notion of counterexample minimization that minimizes values with respect to the type system of the language being model checked, rather than at the level of SAT variables. Greedy and optimal approaches to the minimization problem are presented and compared. The second contribution extends a BMC-based error explanation approach to automatically hypothesize causes for the error in a counterexample. These hypotheses (in terms of relationships between variables) can be automatically checked to determine if a causal dependence exists. Experimental results show that causes can be automatically determined for errors in interesting ANSI C programs.

*Key words:* model checking, counterexamples, error explanation

## 1 Introduction

The value of counterexamples [9] in model checking [10] is indisputable: Bounded Model Checking (BMC) [4] can even be seen as a recognition of the centrality of the search for a counterexample to a property. For practical purposes,

[1] This research was sponsored by the Gigascale Systems Research Center (GSRC) under contract no. 9278-1-1010315, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of GSRC, NSF, ONR, NRL, ARO, GM, or the U.S. government.

[2] Email: agroce@cs.cmu.edu

[3] Email: kroening@cs.cmu.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  void f (int a, int b, int c)
2  {
3    int temp;
4    if (a > b) {
5      temp = a;
6      a = b;
7      b = temp;
8    }
9    if (b > c) {
10     temp = b;
11     b = c;
12     c = temp;
13   }
14   if (a < b) {
15     temp = a;
16     a = b;
17     b = temp;
18   }
19   assert ((a <= b) && (b <= c));
20 }
```

Fig. 1. sort.c

the diagnostic uses of counterexamples in understanding and correcting errors in a system or its specification are obvious.

A model checking counterexample is intended to be read by a person and used for debugging. Ideally, such a counterexample would be the most succinct and easily comprehensible witness to the existence of an error. The utility of small (in various senses, including length) counterexamples is widely recognized. Minimization of counterexamples, both in bounded [23] and explicit-state [14] model checking is a topic of ongoing research.

Previous work on minimization of counterexamples has concentrated either on producing counterexamples of minimal length or on removing irrelevant information from a counterexample. This paper presents a technique that can be used to minimize counterexample length, but focuses on a *semantic* minimization, with respect to the type system of the language (ANSI C, in this case). In particular, this approach minimizes the *values* of variables in the counterexample. As an example, consider the program in Figure 1. Without minimization, the Bounded Model Checker CBMC [19] produces the counterexample shown in Figure 2.

CBMC and similar tools are likely to produce counterexamples with unusually high (or low) values for variables. Bounded model checkers unwind a transition system to produce a propositional formula that is satisfiable if a counterexample of a certain length exists. The SAT solvers used to check these formulas for satisfiability typically return the first satisfying assignment produced. The counterexample values, therefore, are highly dependent on the decision heuristics used by the SAT solver. That these choices may result in needlessly large values for the actual program variables is independent of the issue of unnecessarily complete assignments addressed by other minimization work [23]. The issue is an artifact of the bit-level translation; however, using an integer-based technique such as UCLID [5] would not preserve the proper bit operation and overflow semantics of ANSI C.

```
Counterexample:
Initial State
----------------------------------------------------
State 1
----------------------------------------------------
  a=-8193 (11111111111111111101111111111111)
State 2
----------------------------------------------------
  b=-402 (11111111111111111111111001101110)
State 3
----------------------------------------------------
  c=-2080380800 (10000011111111111110100010000000)
State 4
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 10 file sort.c line 10 function c::f
----------------------------------------------------
  temp=-402 (11111111111111111111111001101110)
State 11 file sort.c line 11 function c::f
----------------------------------------------------
  b=-2080380800 (10000011111111111110100010000000)
State 12 file sort.c line 12 function c::f
----------------------------------------------------
  c=-402 (11111111111111111111111001101110)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 2. Counterexample for sort.c

```
Counterexample:
Initial State
----------------------------------------------------
  temp=-1 (11111111111111111111111111111111)
  a=0 (00000000000000000000000000000000)
  b=0 (00000000000000000000000000000000)
  c=-1 (11111111111111111111111111111111)
State 6 file sort.c line 10 function c::f
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 7 file sort.c line 11 function c::f
----------------------------------------------------
  b=-1 (11111111111111111111111111111111)
State 8 file sort.c line 12 function c::f
----------------------------------------------------
  c=0 (00000000000000000000000000000000)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 3. Minimized counterexample for sort.c

In this case, the decision heuristics used by ZChaff [22] assign 1 to a large number of bits. This results in large values for the program variables, making it difficult to follow what is happening. This problem, already evident in a small example program, is greatly exacerbated when many variable values are involved.

Using the optimization approach to value minimization presented in Section 4.2, we produce a new counterexample with minimal values for the program variables, making it much easier to follow the behavior of the program (Figure 3). Note that both counterexamples are of the same length and contain the same amount of program state.

This paper presents two approaches to value minimization. The first is a greedy approach that makes use of incremental SAT (Section 4.1), while

the second solves an optimization problem in order to guarantee true minimality (Section 4.2). Both approaches are used for counterexample length minimization, as well.

The second issue addressed in this work is that of making better use of counterexamples. Minimization directly improves the usability of counterexamples. *Error explanation* [15] provides information about the causality of errors beyond that contained in the counterexample alone. The `explain` tool [16] automatically generates explanations for CBMC counterexamples, based on the *counterfactual* theory of causality proposed by David Lewis [21]. Previous work [15] presented a notion of *causal dependence*, and noted that `explain` could check whether an error was causally dependent on a predicate. This feature was of limited utility, however, as the user was required to supply a possible cause to be checked. Section 6 presents a new method for automatically hypothesizing possible causes for an error.

## 2 Related Work

Minimization of counterexample length has been addressed in various contexts, including heuristic approaches [9,14,13]. Other kinds of minimization, based on game-semantics or minimal SAT assignments [18,23] have also appeared. The approach presented here for minimizing counterexample values can also be used to minimize counterexample lengths.

More generally, maximizing the utility of counterexamples has been addressed by the ideas of proof-like and evidence-based counterexamples [8,26].

The automatic causal dependence hypothesis-generation and checking presented in Section 6 is a natural extension of BMC-based *error explanation* [15,16]. Error explanation facilities have been described for MSR's SLAM model checker [3] and NASA's JPF model checker [17]. The game-based minimization approach of Jin, Ravi, and Somenzi [18] also provides an error explanation. The distance metric based approach used in Section 6 is related to Zeller's delta-debugging techniques [28,27] and the fault localization approach taken by Renieris and Reiss [24].

## 3 Bounded Model Checking for C Programs

CBMC reduces the model checking problem to determining the validity of a bit-vector equation; full details are presented elsewhere [11]. In a process analogous to that used for Bounded Model Checking of Kripke structures, the transition system is unwound by duplicating the loop bodies in the case of `for` and `while` loops, duplicating code in the case of loops build by means of backward `goto` statements, and function inlining in the case of recursive functions. Unwinding assertions ensure that sufficient unwinding has been performed – i.e. that it is not possible that a counterexample can be found by allowing more loop iterations.

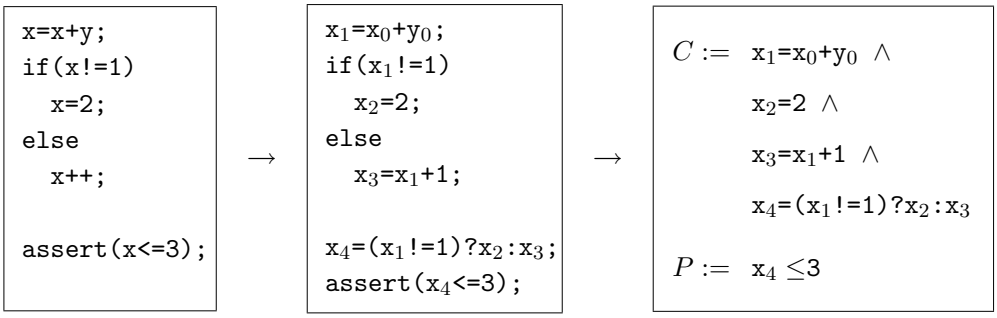


Fig. 4. Transformation into SSA

The program, after unwinding, is composed of only `if` statements and assignments. This program is then transformed into static single assignment (SSA) form [2], which requires a pointer analysis. We omit the full details of this process. For programs in SSA form, each variable is assigned at most once (Figure 4). The SSA form is then transformed into an equation $C$ by replacing the assignments by equalities. The property is denoted by $P$. In order to check the property, CBMC converts $C \wedge \neg P$ into CNF by adding intermediate variables and passes the CNF to a SAT solver such as Chaff [22]. If the equation is satisfiable, the solution to $C$ represents a counterexample for $P$. If it is unsatisfiable, $P$ holds.

The conversion of most operators into CNF is straight-forward and resembles the generation of appropriate arithmetic circuits. The tool can also output the bit-vector equation before it is flattened to CNF, for the benefit of circuit-level SAT solvers.

## 4 Counterexample Value Minimization

### *4.1 Greedy Minimization*

The first method for value minimization is a greedy heuristic approach based on incremental SAT. The first step of the algorithm is to attempt to minimize the length of the counterexample. CBMC generates a Boolean guard variable for each basic block. The variable is 1 if and only if the block is executed in the trace. In hardware BMC, length minimization is not generally an issue if BMC is performed in an incremental fashion. The unwinding used by CBMC, however, includes program statements that may or may not be executed – the unwinding length is *not* the number of execution steps – it is the *potential* number of execution steps, *if* all guards are satisfied, which typically is not the case. Different counterexamples with the same unwinding depth may execute varying numbers of program statements.

The length minimization algorithm sorts the list of guard variables by the number of instructions each guard affects. Starting with the guard that affects the most instructions, the heuristic proceeds as follows: first, a clause is added to the clause data base with the negation of the guard variable as the only

literal (forcing the guard to be false). The algorithm then proceeds depending on the value of the variable in the current satisfying assignment:

- If the value of the variable in the current satisfying assignment is 0, the old satisfying assignment is also a satisfying assignment for the new set of clauses.
- If the value is 1, the SAT solver is restarted. If the new instance is also satisfiable, the new clause remains in the clause database. If it is unsatisfiable, the attempt failed and the new clause is removed.

The algorithm continues with the next guard from the sorted list until all guards have been used. The heuristic approach only then attempts to minimize the variables that are used in the counterexample trace. Because the guard values are now fixed, the only values minimized are those that will appear in the counterexample: assignments guarded by false conditions are not taken into account. Alternatively, one could attempt to minimize first values and then execution steps.

The heuristic begins with the most significant bits of all variables, and then continues towards the least significant bit. In case of unsigned variables, the heuristic attempts to make all the bits zero. Signed variables are encoded as two's complement, and the goal is to minimize the absolute value. Furthermore, positive values are preferred over negative values. Thus, in case of signed variables, the algorithm first tries to set the sign bit to 0. The following bits are then minimized to 0 or 1, depending on the outcome of the SAT instance for the corresponding sign bit. If the sign bit is 1, the heuristic attempts to make the following bits 1 as well, and vice versa.

### *4.2 Optimal Minimization*

The greedy approach to minimization does not always work well. A very unfortunate choice for the initial value to minimize for the program in Figure 1 produces a counterexample (Figure 5) that is not only almost unminimized, but is *longer* than the original counterexample. [4]

True minimization of counterexample values can be considered as a 0-1 ILP problem. PBS [1] is a *pseudo-Boolean* constraint solver which, given a SAT instance in CNF and a set of integer coefficients for SAT variables, will solve optimization problems over the constraints. The length of the counterexample is minimized before values are minimized. Each guard bit is given a weight equal to the number of program statements guarded by that condition. The psuedo-Boolean optimization problem is to minimize the weighted sum, i.e., the number of executed program statements. As with the greedy algorithm, counterexample length minimization is completed and guard values are locked before value minimization begins.

[4] The length increase actually results from the attempt to greedily minimize counterexample length, rather than values.

```
Counterexample:
Initial State
----------------------------------------------------
State 1
----------------------------------------------------
  a=2114977792 (01111110000100000000000000000000)
State 2
----------------------------------------------------
  b=-33554433 (11111101111111111111111111111111)
State 3
----------------------------------------------------
  c=2138989455 (01111111011111100110001110001111)
State 4
----------------------------------------------------
  temp=0 (00000000000000000000000000000000)
State 6 file sort.c line 5 function c::f
----------------------------------------------------
  temp=2114977792 (01111110000100000000000000000000)
State 7 file sort.c line 6 function c::f
----------------------------------------------------
  a=-33554433 (11111101111111111111111111111111)
State 8 file sort.c line 7 function c::f
----------------------------------------------------
  b=2114977792 (01111110000100000000000000000000)
State 14 file sort.c line 15 function c::f
----------------------------------------------------
  temp=-33554433 (11111101111111111111111111111111)
State 15 file sort.c line 16 function c::f
----------------------------------------------------
  a=2114977792 (01111110000100000000000000000000)
State 16 file sort.c line 17 function c::f
----------------------------------------------------
  b=-33554433 (11111101111111111111111111111111)
Failed assertion: assertion file sort.c line 19 function c::f
```

Fig. 5. Greedily minimized counterexample for sort.c

The notion of value minimality used here is to minimize the sum of the absolute values of all variables (appearing in the counterexample), with respect to the type system of the language. Again, consider the program in Figure 1. At first glance, it would appear that the goal is to minimize the sum: `|a| + |b| + |c| + |temp|`. However, each of these variables may take on different values during execution of the program. Therefore, the sum that is minimized is over all program variables after loop unrolling and static single assignment [2], in this case `|a#0| + |a#1| + |a#2| + |a#3| + |a#4| + |b#0| + ... + |b#6| + |c#0| + |c#1| + |c#2| + |temp#0| + ... + |temp#6|`.

For unsigned bit-vectors, the pseudo-Boolean constraints produced simply use values proportional to the place values, i.e., the least significant bit receives a weight of 1, the next least significant receives a weight of 2, up to a weight of $2^{n-1}$ for the most significant bit of an $n$-bit vector. Let $a_0$ denote the least significant bit of the bit-vector $a$, and $a_{n-1}$ the most significant bit. We denote the integer value of an unsigned bit-vector $a$ by $\langle a \rangle$:

$$\langle a \rangle := \sum_{i=0}^{n-1} a_i \cdot 2^i \quad (1)$$

For signed bit-vectors, such as those in `sort.c`, a different approach is required. CBMC encodes signed bit-vectors using two's complement. Note

that the ANSI C standard also permits other encodings.

The integer value represented by $a$ is in the range from $-2^{n-1}$ to $2^{n-1}-1$ and is denoted by $[a]$:

$$[a] := -2^{n-1} \cdot a_{n-1} + \langle a_{n-2} \ldots a_0 \rangle \quad (2)$$

The bit $a_{n-1}$ is called the sign bit. We denote it by $sign(a)$.

We aim at minimizing the absolute value of $a$. If $a$ is positive, i.e., if $a_{n-1}$ is 0, then the absolute value of $a$ is equal to $a$. If the sign bit is 1, the negation of $a$ is

$$-[a] = \langle \overline{a} \rangle + 1 \quad (3)$$

where $\overline{a}$ denotes the bit-wise negation of $a$.

We implement this computation as follows: For each variable $x$ of a signed bit-vector type, we introduce a new variable $x'$ which is the bit-wise xor of $x$ with its own sign bit:

$$x'_i := x_i \oplus sign(x) \quad (4)$$

If $x$ is positive, then $x' = x$ and it is obvious that $x'$ it is equal to the absolute value of $x$, i.e.,

$$|[x]| \; = \; \langle x' \rangle$$

If $x$ is negative, then $x' = \overline{x}$, and obviously

$$|[x]| \; = \; -[x] \; = \; \langle x' \rangle + 1$$

Combining both cases results in

$$|[x]| \; = \; \langle x' \rangle + x_{n-1}$$

As $x'_{n-1}$ is always zero, we get

$$|[x]| \; = \; \langle x'_{n-2} \ldots x'_0 \rangle + x_{n-1}$$

The pseudo-Boolean constraints are then assigned almost exactly as in the unsigned case, with the following exceptions: (i) the constraints for every place value, *not including the sign bit* are based on $x'$ rather than $x$ and (ii) the weight of the sign bit is 1, rather than $2^{n-1}$. Thus, we minimize

$$x_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot x'_i \quad (5)$$

## 5 Experimental Results

Table 1 shows results for minimization of counterexamples for several programs. The first column shows which program is being model checked. The remaining columns give results for the non-minimized counterexample, the greedily minimized counterexample, and the optimally minimized counterexample, in groups of three. In each group, the first column gives the time taken to generate a counterexample (time(s)). The second column ($\Sigma[x]$) in each group gives the sum of the absolute values of the variables and the third

| | normal | | | greedy | | | optimal | | |
|---|---|---|---|---|---|---|---|---|---|
| Program | time(s) | $\Sigma[x]$ | $l$ | time(s) | $\Sigma[x]$ | $l$ | time(s) | $\Sigma[x]$ | $l$ |
| sort.c | 0.70 | $4.161 \times 10^9$ | 7 | 1.11 | $1.073 \times 10^{10}$ | 10 | 995.72 | 2 | 7 |
| TCAS #1 | 1.30 | 111,905 | 73 | 5.22 | 111,905 | 73 | 13.35 | 9,734 | 73 |
| TCAS #11 | 1.20 | 747,623 | 65 | 4.44 | 747,623 | 65 | 14.55 | 9,524 | 65 |
| TCAS #31 | 1.64 | 488,241 | 68 | 4.05 | 488,241 | 68 | 12.23 | 8,932 | 68 |
| TCAS #40 | 0.87 | 640,307 | 63 | 3.83 | 640,307 | 63 | 5.32 | 9,526 | 63 |
| TCAS #41 | 1.69 | 937,749 | 72 | 4.00 | 937,749 | 72 | 6.05 | 9,528 | 72 |
| adpcm_coder | 4.42 | 814 | 106 | 39.52 | 73 | 91 | 107.30 | 391 | 91 |
| adpcm_decoder | 2.47 | 578 | 83 | 41.20 | 517 | 78 | 9.49 | 574 | 73 |
| epic_quantize | 1.06 | 18 | 28 | 7.58 | 14 | 28 | 3.65 | 14 | 28 |
| g721_decode | 8.10 | $1.075 \times 10^9$ | 289 | 168.63 | 855,224 | 298 | 18.45 | 855,106 | 289 |
| gsm_decode | 367.60 | 5257 | 250 | 3667.68 | 3,166 | 374 | 2436.08 | 180,041 | 225 |
| mpeg2dec | 3.82 | $6.334 \times 10^9$ | 61 | 141.41 | 9 | 60 | 55.36 | 9 | 60 |

Table 1
Time and minimization results for greedy and optimal strategies.

column ($l$) gives the length in steps of the counterexample. Sums over 1 billion are given in scientific notation. The benchmarks are taken from the example program `sort.c`, the TCAS suite [25], and the MediaBench benchmarks [20].

For the sort example and TCAS benchmarks, greedy optimization resulted in no improvements in the original counterexamples but in all cases took less time than true optimization.

For the MediaBench benchmarks, the results are mixed. The greedy heuristic is typically slower than the true optimization, but results in smaller values in some cases (the values are a secondary goal, and larger values in the optimal algorithm can be caused by different control flow traces computed in the first stage). On two benchmarks, hardly any minimization is achieved by either algorithm. These benchmarks make heavy use of large lookup-arrays, which are computed at run-time.

## 6 Hypothesizing and Checking Causal Dependence

Previous work [15] using CBMC to explain errors in programs presented a notion of *causal dependence* derived from David Lewis' counterfactual theory of causality [21]:

**Definition 6.1** [causal dependence] A predicate $e$ is *causally dependent* on a predicate $c$ in an execution $a$ iff:

(i) $c$ and $e$ are both true for $a$ (we abbreviate this as $c(a) \wedge e(a)$)

(ii) $\exists$ an execution $b$ . $\neg c(b) \wedge \neg e(b) \wedge (\forall b' \,.\, (\neg c(b') \wedge e(b')) \Rightarrow (d(a,b) < d(a,b')))$

where $d$ is a *distance metric* for program executions. In other words, $e$ is causally dependent on $c$ in an execution $a$ iff executions in which the removal

of the cause also removes the effect are more like $a$ than executions in which the effect is present without the cause.

The previous work did not focus on checking causal dependence, as determining if $e$ depends on $c$ is only useful *after* arriving at likely candidate causes. This would be putting the cart before the horse, as the chief goal of error explanation is to help the user move from awareness of the existence of an error to a small set of candidate causes. The distance metric that allows causal dependence checking was instead used to discover a successful execution that was as similar as possible to a given counterexample. The distance metric used by CBMC is based on the total number of atomic changes ($\Delta$s) in variable and guard values between two executions [15]. These differences are presented to the user as causes for the error.

However, differences in actual variable values are often *too specific*. The relevant information is often a change in *relationships* between variables: i.e., not that `x` was 100 and must be changed to 200 to avoid violating an assertion, but that in the failing run `x < y` and in the successful run, `x > y`. The basic explanation approach may, unfortunately, completely omit `y` from an explanation if only the value of `x` is altered in the successful execution. Because the distance metric minimizes the number of changes, such omissions are very likely to occur. A more general notion of $\Delta$s would report to the user all predicates whose values are different for the counterexample and the successful execution. As the set of changed predicates is potentially infinite (comparisons of variables with constant values, etc.), only a subset of the potential $\Delta$s can be considered. Our implementation only checks basic ordering and equality relations between program variables, e.g. `x == y`, `x < y`, `x > y`, `x <= y`, etc.

Directly presenting the set of changed $\Delta$s is not particularly useful: changes in important variables are likely to introduce many accidental and unimportant changes, hiding the relevant differences in a large set of uninteresting results. However, the set of changes can be used a set of *candidate causes* for *checking causal dependence*. Only the $\Delta$s on which the error is causally dependent are presented to the user.

The set of predicate $\Delta$s that need to be checked is minimized by requiring that one of the variables being compared has changed its value in the successful execution. If neither variable has changed value, the predicate value must be unchanged. Given a possible cause $c$, the counterexample execution $a$, and an error (or effect) $e$, checking causal dependence requires two steps:

(i) Find an execution $b$ such that (1) $c$ does not hold and (2) the distance $d(a, b)$ is minimal. $b$ is an execution that is as similar as possible to the counterexample $a$, except that the potential cause $c$ is present in $a$ but not in $b$. If the error $e$ is present in $b$, it is not causally dependent on $c$.

(ii) Perform bounded model checking over all executions such that (1) $c$ does not hold and (2) the distance to $a$ is equal to $d(a, b)$. If all such executions are error free ($e$ does not hold), then $e$ is causally dependent on $c$.

```
Error is causally dependent on these predicates:
  c#0 < a#0
  c#0 < b#0
```

Fig. 6. Causes for sort.c

```
Error is causally dependent on these predicates:
  Input_Down_Separation#0 == Layer_Positive_RA_Alt_Thresh#1
  Input_Down_Separation#0 <= Layer_Positive_RA_Alt_Thresh#1
  Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1
  Down_Separation#1 <= Layer_Positive_RA_Alt_Thresh#1
```

Fig. 7. Causes for TCAS error #1

```
 100c100
// (correct version)
<       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation >= ALIM())));
---
// (faulty version #1)
>       result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          (!(Down_Separation > ALIM())));
```

Fig. 8. diff of correct TCAS code and variation #1

Figure 6 shows a subset of the causes discovered for the counterexample shown in Figure 3. In this case, the only causes shown are those which relate two input values. The algorithm actually detects 63 additional causes, relating inputs to intermediate values, or intermediate values to each other. For this reason, an option is provided to only check for relationships between input variables. The high degree of causal dependence in this case derives from the nature of the code: for a faulty sorting routine, ordering relations will obviously be crucial to the occurrence of the error, unless the sorting routine is invariably incorrect. The relationships between intermediate values are somewhat uninteresting in this case, as the set of input values is equivalent to the set of all values computed by the program.

For variation # 1 of the TCAS case study [25,12] examined in earlier work [15,16], however, a much smaller set of causes (Figure 7) is produced without restriction to input values. Figure 8 shows the error in the TCAS code as a diff between correct and incorrect versions. The automatically generated explanation, as described in the earlier work, focuses attention on line 100. The function call to `ALIM()` on this line always returns a value that is equal to `Layer_Positive_RA_Alt_Thresh#1`. Any user familiar with the specification of the TCAS code will be aware of this equivalence. Knowing (i) that the fault can be localized to line 100 and (ii) that the error is causally dependent on the predicate `Down_Separation#1 == Layer_Positive_RA_Alt_Thresh#1`, a user should be able to quickly conclude that the `>` comparison on line 100 should be a `>=` comparison.

### *6.1 Alternative Approaches for Hypothesis Selection*

The particular choice of predicates for which to check causal dependence involves a number of tradeoffs: using too many predicates will increase com-

putation time and may result in redundant results; using too few may miss causal dependencies. An obvious alternative method is to use predicates taken from guards and Boolean assignments in the program source. Such comparisons should be generalized: if `x > y` appears in a guard, checking `x <= y`, `x == y`, and so forth is necessary to catch cases where the choice of comparison operations is incorrect. The primary difference between this generalization and the method implemented in CBMC is that no causality checking is done for (1) comparisons with constants and (2) comparisons with temporary results that are never stored in a variable (i.e. `x > (y + 50)`) are not checked for causality. On the other hand, comparisons between values that do not appear in guards together are checked. Causal dependencies that are directly present in a guard in the source code are generally not as difficult to detect as indirect dependencies: a change in guard value is likely to appear in the explanation. For this reason, it seems at least reasonable to expect that the current tradeoff is often the correct choice. More extensive evaluation will be needed to determine if a source code-mining approach is preferable.

Another alternative approach is to leverage predicate abstraction. The predicate abstraction based model checker MAGIC [6] now supports distance metric based explanation over *abstract* executions [7]. The predicates used in the abstract model could be tested for causal dependence. Checking causal dependence is less important in this case, however, as the explanations are presented in terms of changes in relationships between variables in the first place, and irrelevant $\Delta$s are suppressed by the metric and the abstract model.

## 7 Conclusions and Future Work

This paper presents a new kind of counterexample minimization: in contrast to previous work, the simplification is with respect to the semantic values of program variables. Small values are particularly beneficial for understanding traces of sequential programs, such as ANSI C programs. Conventional BMC implementations suffer from the fact that SAT solvers choose values according to built-in heuristics which do not favor readable counterexamples.

Two approaches are described: a greedy minimization heuristic using incremental SAT, and an algorithm that computes an exact solution using a pseudo-Boolean solver. The experimental results show that the optimal approach not only produces better results in a great many cases, but that it can be *faster* than the greedy approach. However, both algorithms are considerably slower than plain BMC without minimization.

More sophisticated heuristic approaches taken from the optimization community should outperform the naive greedy implementation. In future work, we plan to investigate SAT solvers with decision heuristics that are aware of a metric for counterexample simplicity: the idea being to make favoring simple counterexamples a part of the search algorithm, as opposed to a post-processing step.

The paper also presents a new use of BMC counterexamples, an extension of previous work on error explanation. This algorithm allows a model checker to identify predicates on which an error is causally dependent, in addition to providing a counterexample and fault localization. In future work, we hope to extend the range of predicates considered and consider subsumption or other methods for reducing the number of causes presented to the user.

## References

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Symposium on the theory and applications of satisfiability testing (SAT)*, pages 346–353, 2002.

[2] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, 1988.

[3] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, 2003.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[5] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification*, pages 78–92, 2002.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[7] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *SIGSOFT/Foundations of Software Engineering*, 2004. To appear.

[8] M. Chechik and A. Gurfinkel. Proof-like counter-examples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, 2003.

[9] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 427–432, 1995.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[11] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using Bounded Model Checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[12] A Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 142–151, 2001.

[13] S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In *Workshop of Software Model Checking (SoftMC)*, 2001.

[14] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in spin. In *SPIN Workshop on Model Checking of Software*, pages 92–108, 2004.

[15] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[16] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In *Computer-Aided Verification*, 2004. To appear.

[17] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[18] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–458, 2002.

[19] D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[20] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[21] D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[23] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, 2004.

[24] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.

[25] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1999.

[26] L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer-Aided Verification*, pages 455–470, 2002.

[27] A. Zeller. Isolating cause-effect chains from computer programs. In *Foundations of Software Engineering*, pages 1–10, 2002.

[28] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

# Bounded Model Checking with SNF, Alternating Automata, and Büchi Automata

Daniel Sheridan [1]

*School of Informatics*
*University of Edinburgh*
*Edinburgh, UK*

**Abstract**

Model checking of LTL formulæ is traditionally carried out by a conversion to Büchi automata, and there is therefore a large body of research in this area, including some recent studies on the use of alternating automata as an intermediate representation.

Bounded model checking has until recently been apart from this, typically using a direct conversion from LTL to propositional logic. In this paper we give a new bounded model checking encoding using alternating automata and focus on the relationship between alternating automata and SNF. We also explore the differences in the way SNF, alternating, and Büchi automata are used from both a theoretical and an experimental perspective.

*Key words:* Bounded model checking, SNF, LTL, Büchi automata, Alternating automata

## 1 Introduction

Before the introduction of bounded model checking in 1999 [1], LTL model checking was typically performed by converting the formula to an automaton expressing the formula, forming the product with the model automaton, then checking the result for emptiness. Research into producing the smallest automaton for a given LTL has been extensive and varied. There is literature giving improvements to the original "GPVW" conversion algorithm [11] including simplifying the LTL before conversion, and the automaton after conversion (eg,[6]) as well as the conversion itself. Some recent work [9,10] proposes the use of alternating automata (AA) as an intermediate representation of the formula. The LTL to AA conversion is linear space so allows for simplifications to be easily performed before the exponential space conversion to a Büchi automaton.

[1] Email: d.j.sheridan@sms.ed.ac.uk

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

Bounded model checking (BMC) has traditionally taken a different approach: the original paper [1] gives an encoding from LTL directly to propositional logic. Being defined recursively on the structure of the formula this (naïvely) appears to be exponential in size, although with careful treatment [4] the result is polynomial in size. An alternative encoding [8] based directly on the fixpoint characterisations of LTL operators produces an encoding which is linear in size. The use of LTL to automata conversions as part of bounded model checking was first explicitly suggested by de Moura et al. [5]. The only experimental comparison [4] is very brief and mainly exercises the LTL simplification available in many automata conversion programs.

Although there are grounds for distinguishing between the direct-to-propositional conversion and the conversions via automata as "syntactic" versus "semantic" [4], we demonstrate in this paper the close correspondence between SNF and alternating automata and their conversion procedures from LTL. We review the use of Büchi automata for BMC and give a new encoding to enable direct use of alternating automata. This allows us to compare more closely the use of the SNF encoding with the use of automata, to explore the advantages and disadvantages of each approach. We demonstrate some of these differences with a series of experiments.

## 2 Background

### *2.1 Bounded model checking*

BMC solves the LTL model checking problem by observing a restricted number of states, $k$. Infinite counterexamples may be represented by a path of the form $ab^\omega$: a *$k$-$l$-loop path* with $k = |ab|$ and $l = |a|$. We constrain a finite sequence of states $\pi$ to be a $k$-$l$-loop by the assertion ${}_lL_k \doteq (\pi(k) = \pi(l))$ [2]. Alternatively we can give finite counterexamples as a *$k$-prefix path* for some LTL properties. In particular, it is not possible to show to give a counterexample for $\mathbf{F}\, f$ for a $k$-bounded path. Typically, we verify a model by examining a sequence of $k$ states $\pi$ interpreted as either a prefix or a loop; we write a disjunction over the $k$ possible interpretations, testing all of the options for the type of path and the value of $l$ simultaneously.

### *2.2 The Separated Normal Form*

SNF [7] is a clause-like normal form based on the Separation Theorem of Gabbay, with the general form $\mathbf{G} \bigwedge_i (P_i \Rightarrow F_i)$ where $P_i \Rightarrow F_i$, called *rules* are restricted to (writing $p$ and $f$ for propositional formulæ)

**Initial rules** of the form $\mathbf{start} \Rightarrow f$ where **start** holds only in the initial state of each path

[2] Note that we give an equivalence between $\pi(k)$ and $\pi(l)$ rather than the transition in the original presentation [1]

**Global invariant rules** $p \Rightarrow f$ with no temporal operator

**Global step rules** $p \Rightarrow \mathbf{X}\, f$

**Global eventuality rules** $p \Rightarrow \mathbf{F}\, f$

Frisch et al. [8] describe a series of transformations from LTL to SNF aimed at bounded model checking. Transformation is linear time and space: an LTL formula with $n$ temporal operators produces up to $2n$ rules and introduces $n$ new variables.

### *2.3 Büchi Automata*

**Definition 2.1** A Büchi automaton $\mathcal{B}$ is defined by the tuple $\langle Q, \Sigma, \delta, I, T \rangle$ where $Q$ is the set of states; $\Sigma$ is the alphabet of transition labels; $\delta$ is the transition function $Q \to 2^{2^{\Sigma} \times Q}$; $I \subseteq Q$ is the set of initial states; $T \subseteq Q$ is the set of accepting states.

Note that we use $2^{\Sigma}$ in the definition of the transition relation in place of $\Sigma$ in order to gather transitions that differ only by their actions — this can be a significant optimisation.

A run of a Büchi automaton is a path through the automaton; it is accepting if the states in $T$ are visited an infinite number of times. That is,

**Definition 2.2** A *run* of a Büchi automaton $\mathcal{B}$ with respect to a word $u_0 u_1 \ldots \in \Sigma^{\omega}$ is a sequence of states in $q_0 q_1 \ldots \in Q^{\omega}$ with $q_0 \in I$ and $\forall i \exists \alpha_i \, \langle \alpha_i, q_{i+1} \rangle \in \delta(q_i)$ such that $u_i \in \alpha_i$. A run is *accepting* if infinitely many states in the run are members of $T$.

A *generalised* Büchi automaton (GBA) has a set of accepting sets $\mathcal{T} \subseteq 2^{Q}$; each set must be visited infinitely often for acceptance. A GBA may be reduced to a classical Büchi automaton but incurs a linear blowup of $O(|\mathcal{T}|)$.

### *2.4 Alternating Automata*

Alternating automata are a type of tree automaton (runs are described as trees rather than linear traces) combining both deterministic and nondeterministic behaviours: a transition in a nondeterministic automaton leads to a set of states from which one is chosen; a transition in a deterministic tree automaton leads to a successor set. Alternating automata exhibit the combination of these existential and universal behaviours. Although the presentation that we adopt below is one of a nondeterministic choice between conjunctions of states, it can be generalised to arbitrary propositional formulæ over $\wedge, \vee$ and states. Alternating automata are exponentially more succinct than Büchi automata.

There are two presentations of LTL to automata conversion via alternating automata. We follow the slightly unconventional presentation by Gastin and Oddoux [10]: transitions are from a state to a conjunction of states; each state may have multiple transitions, selected nondeterministically. This effectively

encodes a disjunction of conjunctions of states reached from a given state. The presentation given by Fritz and Wolper [9] is equivalent, but the differences in the definitions lead to larger representations of the automata.

**Definition 2.3** An *alternating co-Büchi automaton* $\mathcal{A}$ is defined by the tuple $\langle Q, \Sigma, \delta, I, F\rangle$ where $Q$ is the set of states; $\Sigma$ is the alphabet of transition labels; $\delta$ is the transition function $Q \to 2^{2^{\Sigma}\times 2^{Q}}$; $I \subseteq 2^{Q}$ is the set of initial combinations of states; $F \subseteq Q$ is the set of final states

As for the Büchi automaton definition above, the transition labels are from $2^{\Sigma}$; accepted words are nevertheless from $\Sigma^{\omega}$.

Alternating automata representing LTL formulæ are known to be *very weak*, which means that there is a partial order on the states $(Q, \sqsubseteq)$ determined by the transitions, such that $\forall q \in Q, \forall \langle \alpha, q'\rangle \in \delta(q), q' \sqsubseteq q$. That is, transitions are only permitted from a state to a lower or equal state. The result of this restriction is that the only loops in very weak co-Büchi alternating automaton (VWAA) are self-loops.

**Definition 2.4** A run $\sigma$ of a VWAA on a word $u_0 u_1 \ldots \in \Sigma^{\omega}$ is a labelled DAG $\langle V, E, \lambda\rangle$ with $V$ partitioned into levels $V_i$, $V = \bigcup_{i\in\mathbb{N}} V_i$ and $E \subseteq \bigcup_{i\in\mathbb{N}} V_i \times V_{i+1}$. $\lambda : V \to Q$ labels the vertices of the graph with states of the automaton. $V_i$ may be seen as a multiset of elements of $Q$. The graph is related to the word and the automaton by $\lambda(V_0) \in I$ and $\forall v \in V_i, \exists \langle \lambda(v), \alpha, s'\rangle \in \delta(\lambda(v)).u_i \in \alpha \wedge s' = \lambda(E(v))$ A run is accepting if every infinite branch of $\sigma$ has only a finite number of nodes with labels in $F$.

### *2.4.1 LTL to VWAA Conversion*

We report here the conversion procedure given by Gastin and Oddoux. The set operator $\otimes$ constructs the conjunctions of two sets of disjunctive normal form transitions: $X \otimes Y = \{\langle \alpha_1 \cap \alpha 2, e_1 \wedge e_2\rangle \mid \langle \alpha_1, e_1\rangle \in X, \langle \alpha_2, e_2\rangle \in Y\}$. The overbar operator $\bar{\psi}$ converts $\psi$ to a set-style disjunctive normal form representation: a set of conjunctions of atomic propositions or temporal subformulæ.

For an LTL formula $\varphi$ over atomic propositions $P$, the VWAA $\mathcal{A}_{\varphi} = \langle Q, \Sigma, \delta, I, F\rangle$ is given by

- $Q$ is the set of temporal subformulæ of Q (the set of subformulæ with an LTL operator as the main connective, union the set of atomic propositions)
- $\Sigma = 2^{P}$; $I = \bar{\psi}$; $F$ is the set of formulæ of the form $\psi_1 \,\mathbf{U}\, \psi_2$ or $\mathbf{F}\,\psi_1$
- $\delta$ is defined as

$$\begin{aligned}
\delta(\top) &= \{\langle \Sigma, \top\rangle\} \\
\delta(p) &= \{\langle \{a \in \Sigma \mid p \in a\}, \top\rangle\} \\
\delta(\neg p) &= \{\langle \{a \in \Sigma \mid p \notin a\}, \top\rangle\} \\
\delta(\mathbf{X}\,\psi) &= \{\langle \Sigma, e\rangle \mid e \in \bar{\psi}\} \\
\delta(\mathbf{F}\,\psi) &= \Delta(\psi) \cup (\{\langle \Sigma, \mathbf{F}\,\psi\rangle\})
\end{aligned}$$

$$\begin{aligned}\delta(\mathbf{G}\,\psi) &= \Delta(\psi) \otimes \{\langle \Sigma, \mathbf{G}\,\psi\rangle\})\\ \delta(\psi_1\,\mathbf{U}\,\psi_2) &= \Delta(\psi_2) \cup (\Delta(\psi_1) \otimes \{\langle \Sigma, \psi_1\,\mathbf{U}\,\psi_2\rangle\})\\ \delta(\psi_1\,\mathbf{R}\,\psi_2) &= \Delta(\psi_2) \otimes (\Delta(\psi_1) \cup \{\langle \Sigma, \psi_1\,\mathbf{R}\,\psi_2\rangle\})\end{aligned}$$

where $\Delta$ is the extension of $\delta$ to include the propositional subformulæ of $\varphi$:

$$\begin{aligned}\Delta(\psi) &= \delta(\psi) && \text{if } \psi \in Q\\ \Delta(\psi_1 \wedge \psi_2) &= \Delta(\psi_1) \otimes \Delta(\psi_2)\\ \Delta(\psi_1 \vee \psi_2) &= \Delta(\psi_1) \cup \Delta(\psi_2)\end{aligned}$$

*2.4.2 Compact Representation of Runs*

The representation of a run of a VWAA as a DAG is problematic as the number of vertices at each level grows without bound. We can reduce the representation of a run by restricting each level to a set rather than a multiset, forming a reduced DAG. We call successive sets *configurations*, $C_i \subseteq Q$. A sequence of configurations over a word $u_0u_1\ldots \in \Sigma^\omega$ is accepting if there exists a set of edges $E$ partitioned into $E_i \subseteq C_i \times C_{i+1}$ such that $\forall q \in C_i \exists \langle \alpha, q'\rangle \in \delta(q).u_i \in \alpha \wedge q' \subseteq E_i(q)$ and every path $q_0q_1\ldots$ such that $q_{i+1} \in E_i(q_i)$ contains only finitely many occurrences of the members of $F$.

This is significantly weaker than the original formulation, but we can show that the languages accepted are equivalent. Firstly, every accepting sequence of configurations $C_i$ with acceptance described by edges $E_i$ may be directly translated into the DAG $\langle \bigcup_{i\in\mathbb{N}} C_i, \bigcup_{i\in\mathbb{N}} E_i, I\rangle$ where $I$ is the identity function on states. In the opposite direction, every accepting DAG can be reduced to an accepting run of configurations given by $C_i = \bigcup_{v\in V_i} \lambda(v)$. We show this sequence is accepting by appealing to an important property of accepting paths: they are both left-append and suffix closed — that is, a suffix of an accepting path is also accepting, as is an accepting path prefixed with a finite number of additional states. This means that the acceptance condition can on configurations can be reduced to the existence of an accepting path from each element of each $C_i$. This is assured by examination of the DAG, since every element of each $V_i$ must be followed by an accepting sequence of edges.

*2.4.3 Superset Property of Runs*

Both formulations of runs describe the minimal elements (or multiset) of states at each point in time, but neither requires that the set consists solely of these elements. We may, without changing the language accepted, replace $C_i$ with a superset of $C_i$ (similarly $V_i$) provided that successive configurations (levels of the tree) can be modified to accommodate the evolution of the extra states while remaining consistent with the definitions of the runs. This is crucial to the encoding described below: we need only constrain the current configuration to be any superset of that described by the transitions.

## 3 Bounded Model Checking Encodings

Having discussed three representations of LTL formulæ suited to model checking we now turn to the way that these representations can be used for bounded model checking. The encoding of Büchi automata was discussed by de Moura et al. [5] as well as Clarke et al. [4]. The use of SNF for bounded model checking was the subjection of a paper by Frisch et al. [8]. The approach that we take here to bring the encodings together is to isolate the components of the encoding of the specification into three parts: that which constrains the path in all cases; that which constrains the path only when it is a finite path prefix; and that which constrains the path only when it is a $k$-loop. The addition of the first constraint to the original approach [1] has the potential to simplify the resulting formula [3] considerably:

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \mathrm{enc}_c(f, k) \wedge \left( \mathrm{enc}_n(f, k) \vee \bigvee_{l=0}^{k} ({}_l\mathrm{L}_k \wedge \mathrm{enc}_l(f, k, l)) \right)$$

where $\mathrm{enc}_c, \mathrm{enc}_n$, and $\mathrm{enc}_l$ denote the common, finite, and loop encodings as described below.

### *3.1 Bounded Model Checking with Büchi Automata*

We present a variation on the encoding of de Moura et al. [5], making explicit the representation of states in order to avoid the overhead of enforcing mutual exclusion on states. In contrast with other presentations, we use generalised Büchi automata: the complexity of checking multiple acceptance sets is much lower than the overhead of conversion to classical Büchi automata.

All paths accepted by a Büchi automaton are infinite — formulæ with finite counterexamples such as $\mathbf{F}\,\phi$ are encoded with a trivial infinite loop. The finite prefix case is therefore never accepting, and we deduce that $\mathrm{enc}_n(f, k) = \bot$.

Given a generalised Büchi automaton representing LTL formula $f$, $\mathcal{B}_f = \langle Q, \Sigma, \delta, I, \mathcal{T} \rangle$, we encode the current state $q \in Q$ as a base two integer in the range $0 \ldots |Q|-1$: there is a one-to-one mapping $\epsilon \subseteq Q \times \{i \mid 0 \leq i < |Q|-1\}$. That is, for each state $i$, we have a set of propositional variables $q_n(i)$, $0 \leq n < \lceil \log_2(|Q|) \rceil$ and we write $[\![q]\!]^i$ for the assertion that the bit pattern $q_0 q_1 \ldots$ is the base two representation of $\epsilon(q)$. For Büchi automata representing LTL, $\Sigma$ is the set of propositions in that model; the encoding of elements $a \in \Sigma$ is given as $[\![a]\!]^i$ as for the standard encoding.

The transition relation is encoded as a set of constraints on the originating

[3] The formula given is derived from the usual BMC formulation as given in Biere et al. [1]. We write $[\![M]\!]_k$ for the encoding of the model, ${}_l\mathrm{L}_k$ for the constraint that the path is a $k$-$l$-loop, but we omit the $\bigwedge_{0 \leq l < k} \neg {}_l\mathrm{L}_k$ non-loop constraint as suggested by [3]

state, target state, and label. If the transition relation is total, we can write

$$T_{\mathcal{B}_f}(i,k) = \bigvee_{\langle s,\alpha,s'\rangle\in\delta} \bigvee_{a\in\alpha} \left( (\llbracket s \rrbracket^i \wedge \llbracket a \rrbracket^i \wedge \llbracket s' \rrbracket^{i+1}) \right)$$

The initial set is encoded directly as a disjunction over members of $I$:

$$I_{\mathcal{B}_f}(k) = \bigvee_{s\in I} \llbracket s \rrbracket^0$$

Finally, we encode the acceptance sets. The Büchi acceptance condition is that each member of $\mathcal{T}$ is visited infinitely often. As we have ruled out finite path prefixes, we know that all paths being considered are of the form $ab^\omega$. If we assert as part of the loop encoding that the corresponding paths in the Büchi automaton follow the same pattern, we can simply require that representatives from each acceptance set appear in the loop (ie, in $b$):

$$F_{\mathcal{B}_f}(k,l) = \bigwedge_{T\in\mathcal{T}} \bigvee_{i=l}^{k} \bigvee_{s\in T} \llbracket s \rrbracket^i$$

Thus we have

$$\begin{aligned}
\mathrm{enc}_c(f,k) &= I_{\mathcal{B}_f}(k) \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{B}_f}(i,k) \\
\mathrm{enc}_n(f,k) &= \bot \\
\mathrm{enc}_l(f,k,l) &= F_{\mathcal{B}_f}(k,l) \wedge \bigwedge_{i=0}^{\lceil \log_2(|Q|)\rceil - 1} q_i(l) \leftrightarrow q_i(k)
\end{aligned}$$

Although the LTL to Büchi automaton conversion is exponential in the size of the formula, the encoding above introduces only a linear number of variables. The resulting formula is linear size in the product of the number of transitions and $k$ except for $F_{\mathcal{B}_f}$ which is quadratic: $O(|\mathcal{T}|k^2)$.

### *3.2 Bounded Model Checking with Alternating Automata*

The encoding of alternating automata is very similar to Büchi automata. Since a run is a sequence of configurations rather than states we use one state variable to represent each state; configurations are then represented by conjunctions of states. Given a VWAA representing LTL formula $f$, $\mathcal{A}_f = \langle Q, \Sigma, \delta, I, F\rangle$, we encode the presence of a a state $q$ in the $i$th configuration by the variable $q(i)$. A configuration is encoded as a conjunction of its members: we write $\llbracket C \rrbracket^i = \bigwedge_{q\in C} q(i)$, with $\llbracket \emptyset \rrbracket^i = \bot$. Note that this constrains the necessary, but not sufficient, members of the configuration, and so

describes the smallest configuration that describes the run as discussed in Section 2.4.3. The targets of transitions can be seen as subsets of configurations and are hence encoded in the same way.

For VWAAs derived from LTL formulæ as above, the transitions are labelled with a set of sets of atomic propositions: the set of permitted assignments to propositions. These can be denoted[4] by a conjunction of literals where $p \wedge q$ denotes $\{a \in \Sigma \,|\, p \in a\} \cap \{a \in \Sigma \,|\, q \in a\}$. We write $[\![\alpha]\!]^i$ for the conjunction of literals representing $\alpha \in 2^\Sigma$ — this is particularly convenient as the implementation of the LTL to VWAA conversion [10] produces these conjunctions directly.

As before, the transition relation is given as a series of constraints

$$T_{\mathcal{A}_f}(i,k) = \bigwedge_{q \in Q} \left( q(i) \rightarrow \bigvee_{\langle \alpha, q' \rangle \in \delta(q)} \left( [\![\alpha]\!]^i \wedge [\![s']\!]^{i+1} \right) \right)$$

and the initial set of configurations is encoded

$$I_{\mathcal{A}_f}(k) = \bigvee_{C_0 \in I} [\![C_0]\!]^0$$

A VWAA run is accepting if no branch contains an infinite occurrence of elements of $F$. This can be assured on a $k$-prefix path if the empty configuration is reached at any point: the *very weak* property means that all successive configurations are also empty and hence no state is visited infinitely often. This also means that we can reduce the check to an empty $k$th configuration: this will hold even if the first empty configuration is before $k$.

$$P_{\mathcal{A}_f}(k) = \bigwedge_{q \in Q} \neg q(k)$$

For the loop case, we cannot simply check for an infinite number of occurrences of the members of $F$ as the co-Büchi condition is on paths through the configuration space. That is, an accepting run could consist of an infinite number of paths each with a finite number of occurrences of an acceptance state. In this case the acceptance state would appear in a configuration within the loop suggesting that the state was visited infinitely often. In fact, we must make use of the *very weak* condition again: the only loops in VWAAs are self-loops, and hence the only paths that visit a state infinitely often must do so by always taking the self-loop transition. By the left-append and prefix closed property of accepting paths, we can deduce that if it is possible to take a non-self-loop transition from an accepting state then that state must be part

[4] See Remark 2 in Gastin and Oddoux [10]

of an accepting path.

$$F_{\mathcal{A}_f}(k,l) = \bigwedge_{q\in F}\bigvee_{i=l}^{k}\Big([\![q]\!]^i \rightarrow \bigvee_{\substack{\langle\alpha,q'\rangle\in\delta(q)\\ q\notin q'}} ([\![\alpha]\!]^i \wedge [\![q']\!]^{i+1})\Big)$$

Thus we have

$$\begin{aligned}\mathrm{enc}_c(f,k) &= I_{\mathcal{A}_f}(k) \wedge \bigwedge_{i=0}^{k-1} T_{\mathcal{A}_f}(i,k)\\ \mathrm{enc}_n(f,k) &= P_{\mathcal{A}_f}(k)\\ \mathrm{enc}_l(f,k,l) &= F_{\mathcal{A}_f}(k,l) \wedge \bigwedge_{q\in Q} q(l) \leftrightarrow q(k)\end{aligned}$$

This encoding produces a linear number of variables in the size of the LTL formula. The resulting propositional formula is linear in the product of the number of transitions and $k$, again except for $F_{\mathcal{A}_f}$ which is quadratic in $k$.

### *3.3 Bounded Model Checking with SNF*

As SNF is a specialisation of LTL we could encode it using the standard BMC method, but we can produce a much better result by considering the structure of rules. Given a set of rules representing an LTL formula $f$, $\Psi_f$, we form the partition $I, P, X, F \subseteq \Psi_f$ according to the type of rule and consider the interpretation of each type:

**Initial rules** $I$ specify initial conditions so are included in $\mathrm{enc}_c$

**Global invariant rules** $P$ are constraints on the configurations of individual states so are also included in $\mathrm{enc}_c$

**Global step rules** $X$ connect states with their successors, very much like a transition relation. Above, we included the transition relation in $\mathrm{enc}_c$ together with a loop condition on its states in $\mathrm{enc}_l$. However, we can simplify this by isolating the common cases at time $< k$ from the boundary cases which distinguish the behaviour of the finite prefix and $k$-loop conditions. These rules are divided between $\mathrm{enc}_c$ for $i < k$, and $\mathrm{enc}_n$ and $\mathrm{enc}_l$ for $i = k$

**Global eventuality rules** $F$ are superficially similar to acceptance conditions but can be interpreted more directly — as in [1]. For a finite prefix, this is simply a disjunction over states; for a $k$-loop of the form $ab^\omega$, evaluating $\mathbf{F}$ in $b$ is equivalent to evaluating it at the start of $b$.

Thus we have

$$\mathrm{enc}_c(f,k) = \bigwedge_{(\mathbf{start}\Rightarrow f)\in I} [\![f]\!]^0 \;\wedge\; \bigwedge_{i=0}^{k}\bigwedge_{(p\Rightarrow f)\in P} [\![p\rightarrow f]\!]^i \;\wedge\; \bigwedge_{i=0}^{k-1}\bigwedge_{(p\Rightarrow \mathbf{X}\,f)\in X} [\![p]\!]^i_k \rightarrow [\![f]\!]^{i+1}$$

$$\mathrm{enc}_n(f,k) = \bigwedge_{(p \Rightarrow \mathbf{X}\, f) \in X} [\![p]\!]^k \rightarrow \bot \;\wedge \bigwedge_{i=0}^{k} \bigwedge_{(p \Rightarrow \mathbf{F}\, f) \in F} \left( [\![p]\!]^i \rightarrow \bigvee_{j=i}^{k} [\![f]\!]^j \right)$$

$$\mathrm{enc}_l(f,k,l) = \bigwedge_{(p \Rightarrow \mathbf{X}\, f) \in X} [\![p]\!]^k \rightarrow [\![f]\!]^l \;\wedge \bigwedge_{i=0}^{k} \bigwedge_{(p \Rightarrow \mathbf{F}\, f) \in F} \left( [\![p]\!]^i_k \rightarrow \bigvee_{j=\min(i,l)}^{k} [\![f]\!]^j \right)$$

As noted above, SNF is linear in the size of the LTL formula; the number of variables in the encoding is therefore linear in the product of $k$ and the size of the formula. The size of the resulting formula is linear in the product of $k$ and the size of the LTL except for the encoding of eventuality rules which are quadratic in $k$.

### *3.3.1 The Fixpoint Form and Counterexample Length*

This quadratic factor is eliminated by the Fixpoint encoding [8], a refinement of SNF in the bounded case which replaces eventuality rules with step rules using the fixpoint characterisation of $\mathbf{F}$ and introduces a boundary condition asserting that the eventuality does not occur after state $k$. Unfortunately, this can lengthen counterexamples in some circumstances as only a single iteration of the loop is considered when evaluating $\mathbf{F}$. This is solved by projecting eventualities evaluated within the loop back to the start of the loop — equivalent to the direct encoding. By the introduction of an untimed intermediate variable this can be achieved in linear space.

## 4 SNF versus Automata

We have examined two established methods of encoding LTL for bounded model checking and introduced a third: the encoding via alternating automata. We now clarify the relationships and relative advantages of the encodings.

### *4.1 SNF and Alternating Automata*

The configuration view of alternating automata makes it apparent that Fixpoint and AA are nearly equivalent. Step rules in SNF/Fixpoint relate states and their successors to the evolved state of the model, while AA transitions which relates states and their successors to the present state of the model. We can project each SNF variable $x$ created during LTL conversion to a VWAA state $\mathbf{X}\, x$: the set of SNF variables is directly related to the members of the configurations of the VWAA. Furthermore, we can show that SNF step rules created from LTL always have atomic antecedents: a necessary condition to relate step rules to transitions.

The boundary condition used in Fixpoint to represent eventualities corresponds to an assertion that $x$ occurs finitely, not infinitely, often. It is introduced for the same states that, in the alternating automaton conversion,

would be in the co-Büchi acceptance set. The difficulty of checking the co-Büchi acceptance condition are sidestepped by the start-of-loop projection introduced in Section 3.3.1. Effectively, all branches of the run are collapsed into one.

In fact, this is the main advantage of SNF over VWAAs: the encoding of the acceptance set is complex and comparatively large for the alternating automaton encoding. There are other advantages: not being a transition system, the variables introduced by SNF are not included in the loopback condition $L_k$, eliminating the need for the empty-configuration assertion in the finite case. This can even reduce slightly the bound at which counterexamples are found. Alternating automata do benefit from the simplification [10] and simulation [9] reductions, some of which do not project directly to SNF; the advantages of these have the potential to outweigh the drawbacks of the encoding.

### *4.2 SNF and Alternating Automata versus Büchi Automata*

Most of the encoding issues discussed above apply equally to Büchi automata, the exception being the acceptance set which is simpler than the alternating case, although still more complex than the Fixpoint case. The biggest drawback for BMC is the requirement for an infinite path. Safety properties with finite counterexamples must still end up in a loop — in both the specification automaton *and the model* which could lengthen the counterexample considerably. In fact, the best choice for simple specifications seems to be the direct encoding: in such a case, the loop constraint could be eliminated altogether.

There are two other loop-related problems with the use of Büchi automata. Firstly, when both the specification and model automata must be in a loop, the length of the loop is the least common multiple of the lengths of the loops in the two automata on their own. This is not an issue for alternating automata because of the weakness property: all loops will be a single state. Secondly, BMC is able to take special advantage of the loopback where a finite counterexample takes the form $ab^i$. For example, consider the word $xx(abb)^\omega$, which is recognised in this form by the specification $F(b \wedge F(a))$ using the direct or SNF encodings, but which must be expanded to $xxabb(abb)^\omega$ to be recognised by the automata-based encodings.

### *4.3 Empirical Results*

To demonstrate some of the differences between the approaches we give a selection of experimental results comparing a variety of BMC encodings. The existing encodings, the original BMC encoding [1] (marked "Orig." in the results), the SNF encoding and its refinement [8] ("SNF" and "FIX") are compared against Büchi automata, in this case the Etessami and Holzmann [6] procedure ("TMP"), and the VWAA produced by the tool from Gastin and Oddoux [10] with and without its simplifications ("AA" and "AA-").

To provide a comparison over a range of LTL specifications we fix the

model for the experiments, using a distributed mutual exclusion example [12] with the specifications given in Frisch et al. [8], at several bounds to illustrate scalability. The number and nesting depths of temporal operators appearing in the specifications are reported as pairs of numbers alongside their names in the tables. We used a modified version of NuSMV [2] with an improved CNF conversion [14]; timings were made in the SAT solver zChaff [13].

The first table, below, shows three correct specifications, verified at bound 50. Rather than report the number of states that each automata conversion produces, we report the size of the CNF result. This means that the automaton methods can be directly compared to the SNF and direct encodings. We observe that as the specifications become more complex, the simplicity of the SNF encoding has an increasing advantage. The alternating automata approach lags close behind the Büchi automata produced by TMP: a particularly interesting result, as the latter includes advanced simulation-based simplification techniques, while the former uses simple transition and state simplifications.

| Enc. | Clauses | Vars | Time | Clauses | Vars | Time | Clauses | Vars | Time |
|---|---|---|---|---|---|---|---|---|---|
| | Accessibility (4,2) | | | Overtaking 1 (5,5) | | | Overtaking 2 (8,8) | | |
| AA | 24276 | 4080 | 4.67 | 26177 | 4131 | 8.45 | 27179 | 4233 | 10.11 |
| AA- | 25485 | 4539 | 5.28 | 28291 | 4845 | 11.87 | 30045 | 5049 | 7.23 |
| SNF | 23778 | 3978 | 4.04 | 24081 | 4080 | 2.76 | 24634 | 4233 | 2.22 |
| FIX | 23779 | 4029 | 4.43 | 24083 | 4131 | 4.17 | 24636 | 4284 | 2.59 |
| TMP | 24279 | 3978 | 4.90 | 27539 | 4029 | 3.54 | 29758 | 4080 | 7.07 |
| Orig | 28368 | 3876 | 9.83 | 142404 | 3876 | 17.69 | Encoding $>$ 1800 secs | | |

We illustrate the effect of the different encodings on counterexample size by comparing two incorrect specifications with different minimal counterexamples (overleaf). Here we see that the Büchi automaton procedure is slower due to the longer counterexample produced. The other procedures are all comparable although the VWAA method is slightly faster on the larger example.

## 5 Conclusions and Future Work

The main advantage of automata based bounded model checking, the high state of development of the conversion procedures, is balanced by the numerous drawbacks of conversion. We have described how the use of alternating automata overcomes many of these problems and demonstrated their use for BMC. A simple alternating automata encoding has been shown to be almost as effective as a highly developed Büchi automata approach, although both lag behind the SNF encoding (without any simplification) on many of the examples given.

| Enc. | $k$ | Time | $k$ | Time |
|---|---|---|---|---|
| | Priority 1 (4,2) | | Priority 2 (4,2) | |
| AA | 14 | 0.03 | 53 | 0.30 |
| AA- | 14 | 0.03 | 53 | 0.89 |
| SNF | 13 | 0.02 | 52 | 0.49 |
| FIX | 13 | 0.02 | 52 | 0.83 |
| TMP | 53 | 3.26 | $> 200$ | |
| Orig | 13 | 0.02 | 52 | 1.15 |

This work has indicated several promising directions for further development. Simulation-based simplification for alternating automata [9] may improve the performance of the approach, and the close relationship with SNF could mean that the SNF encoding could also be improved by such simplification techniques. This relationship could also yield better encodings for the co-Büchi condition, further improving the performance.

## 6 Acknowledgements

I would not have begun to investigate these encoding methods without the helpful comments from the reviewers for CHARME 2003. I am also indebted to Dr Paul Jackson for extensive discussion and input on the topics in this paper.

## References

[1] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, Lecture Notes in Computer Science **1579** (1999), pp. 193–207.

[2] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri, NuSMV*: a new Symbolic Model Verifier*, in: N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science (1999), pp. 495–499.

[3] Cimatti, A., M. Pistore, M. Roveri and R. Sebastiani, *Improving the encoding of LTL model checking into SAT*, in: A. Cortesi, editor, *Third International Workshop on Verification, Model Checking and Abstract Interpretation*, Lecture Notes in Computer Science **2294** (2002), pp. 196–207.

[4] Clarke, E. M., D. Kroening, J. Ouaknine and O. Strichman, *Completeness and complexity of bounded model checking*, in: B. Steffen and G. Levi, editors,

*Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, Lecture Notes in Computer Science **2937** (2004), pp. 85–96.

[5] de Moura, L., H. Rueß and M. Sorea, *Lazy theorem proving for bounded model checking over infinite domains*, in: A. Voronkov, editor, *Automated Deduction - CADE-18; 18th International Conference on Automated Deduction*, Lecture Notes in Computer Science **2392** (2002), pp. 438–455.

[6] Etessami, K. and G. J. Holzmann, *Optimizing Büchi automata*, in: C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, Lecture Notes in Computer Science **1877** (2000), pp. 153–167.

[7] Fisher, M., *A resolution method for temporal logic*, in: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)* (1991), pp. 99–104.

[8] Frisch, A., 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; 4th International Conference, FMCAD 2002*, Lecture Notes in Computer Science **2517** (2002), pp. 238–254.

[9] Fritz, C., *Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata*, in: O. H. Ibarra and Z. Dang, editors, *Implementation and Application of Automata. Eighth International Conference (CIAA 2003)*, Lecture Notes in Computer Science **2759**, Santa Barbara, CA, USA, 2003, pp. 35–48.

[10] Gastin, P. and D. Oddoux, *Fast LTL to Büchi automata translation*, in: G. Berry, H. Comon and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science (2001), pp. 53–65.

[11] Gerth, R., D. Peled, M. Vardi and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in: P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, IFIP Conference Proceedings **38** (1995), pp. 3–18.

[12] Martin, A. J., *The design of a self-timed circuit for distributed mutual exclusion*, in: H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI* (1985), pp. 245–260.

[13] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *39th Design Automation Conference*, Las Vegas, 2001, pp. 530–535.

[14] Sheridan, D., *The optimality of a fast CNF conversion and its use with SAT*, Technical Report APES-82-2002, APES Research Group (2004), available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.

# Authors