First International Workshop on Bounded Model Checking

BMC'03

13 July, 2003

Boulder, Colorado, USA

Preface

This binder includes the preliminary proceedings of the first international workshop on Bounded Model Checking (BMC'03) that was held on July 13th, 2003 in Boulder, Colorado. The final proceedings will be published in issue 4, Volume 89 of Electronic Notes in Theoretical Computer Science (ENTCS), together with other CAV'03 affiliated workshops.

Since its introduction in 1999, Bounded Model Checking has been adopted by most relevant companies as a complementary technique to the more traditional BDD-based unbounded symbolic model checking. Largely due to the advances in SAT technology in the last few years, it became a leading tool in detection of relatively shallow logical errors, outperforming BDD based tools in most of these cases. The large interest in this technology has created a constant stream of new ideas and improvements that make this technique more and more useful. It also led to an effort, reported in the invited talk of this workshop, to use the power of SAT solvers for standard, i.e., unbounded, model checking.

The aim of the workshop was to provide a forum for presenting new results, both theoretical and experimental, in Bounded Model Checking. This is the first workshop to concentrate on this topic, and we hope that it will be followed by similar meetings annually.

Each of the papers selected to this workshop has been reviewed and recommended by at least three (typically four) program committee members. We thank the program committee members for their effort in evaluating the articles. We also thank the organizers of the hosting conference (CAV'03), W. Hunt and F. Somenzi.

Ofer Strichman
Carnegie-Mellon University

Armin Biere
ETH Zürich

July 2003

Organizers

Ofer Strichman	(Carnegie-Mellon University, USA)
Armin Biere	(ETH Zürich, Switzerland)

Program committee

David Basin	(ETH Zürich, Switzerland)
Armin Biere	(ETH Zürich, Switzerland)
Per Bjesse	(Synopsys, USA)
Alessandro Cimatti	(IRST, Italy)
Raanan Fraer	(Intel, Israel)
Danny Geist	(IBM - HRL, Israel)
Alan Hu	(Univ. of British Columbia, Canada)
James Kukula	(Synopsys, USA)
Ken McMillan	(Cadence, USA)
Sharad Malik	(Princeton Univ., USA)
Mary Sheeran	(Chalmers Univ. of Technology, Sweden)
Joao M. Silva	(Technical Univ. of Lisbon, Portugal)
Ofer Strichman	(Carnegie Mellon Univ., USA)
Toby Walsh	(Univ. of York, UK)
Yunshan Zhu	(Synopsys, USA)

Program

Invited talk

Ken McMillan
From Bounded to Unbounded Model Checking

Lectures

Niklas Eén, Niklas Sörensson
Temporal Induction by Incremental SAT Solving

Toni Jussila, Keijo Heljanko, Ilkka Niemelä
BMC via On-the-Fly Determinization

Parthasarathy Madhusudan, Wonhong Nam, Rajeev Alur
Symbolic Computational Techniques for Solving Games

Zurab Khasidashvili, Ziyad Hanna
SAT-based Methods for Sequential Hardware Equivalence Verification Without Synchronization

Bing Li, Fabio Somenzi
A Satisfiability-Based Approach to Abstraction Refinement in Model Checking

Gianpiero Cabodi, Alex Kondratiev, Sergio Nocco, Stefano Quer, Yosinori Watanabe
A BMC-Formulation for the Scheduling Problem in Highly Constrained Hardware Systems

Temporal Induction by Incremental SAT Solving

Niklas Eén, Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract

We show how a very modest modification to a typical modern SAT-solver enables it to solve a series of related SAT-instances efficiently. We apply this idea to checking safety properties by means of *temporal induction*, a technique strongly related to *bounded model checking*. We further give a more efficient way of constraining the extended induction hypothesis to so called *loop-free* paths. We have also performed the first comprehensive experimental evaluation of induction methods for safety-checking.

1 Introduction

In recent years, SAT-based methods for hardware verification have become an important complement to traditional BDD-based model checking. Several methods have proven their usefulness on a number of industrial applications, in particular *bounded model checking* (BMC) [BCCZ99,BCRZ99,CFF+01]. In this paper we will focus our attention on how SAT-based verification procedures can be implemented more efficiently by a tighter integration with the underlying SAT-solver.

There are three main contributions of the paper. Firstly, we show how a number of similar SAT-instances can be solved incrementally by a very modest modification of a modern Chaff-like SAT-solver [MZ01]. The technique we propose is simpler than previous attempts [WKS01], while still obtaining a performance increase of the same magnitude. Secondly, we demonstrate the incremental technique on *temporal induction* [SSS00], a method of checking safety properties on *finite state machines* (FSM). We show the impact of the incremental approach experimentally, both for proving correctness and for finding counter-examples.

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

Thirdly, we refine the method of ensuring completeness for temporal induction. The standard method works by requiring all states in the induction hypothesis to be *unique*. By a simple analysis of the FSM, we are able to exclude some state-variables from the uniqueness constraints, resulting in stronger requirements. This may exponentially reduce the induction depth needed. We prove that this strengthening is sound. Additionally, we demonstrate a speed-up by adding the unique states requirement dynamically for only those pairs of states where it is needed.

The experiments we have performed with our prototype tool Tip show that many properties can be proven at speeds comparable to mature BDD-based tools such as Cadence SMV and CMU SMV.

2 Preliminaries

In this paper, we consider *safety properties* on *finite state machines* (FSM). The states of the FSM are vectors of booleans, defining the values of the *state variables*. We assume the FSM to have a set of legal *initial states*, and the safety property to be specified as a propositional formula over the state variables. By *reachable state space* we mean all states of the FSM reachable from the initial states. Our task is to prove that the property holds for each state in the reachable state space.

In a standard manner, we will assume the transitions of the FSM to be represented by a propositional formula $\mathbf{T}(\boldsymbol{s}, \boldsymbol{s}')$, the set of initial states by a formula $\mathbf{I}(\boldsymbol{s})$, and further denote the safety property by $\mathbf{P}(\boldsymbol{s})$. We will use $\boldsymbol{s_n}$ to denote the state variables of time step n and introduce the shorthand notation $\mathbf{I}_n$, $\mathbf{P}_n$, and $\mathbf{T}_n$ for $\mathbf{I}(\boldsymbol{s_n})$, $\mathbf{P}(\boldsymbol{s_n})$, and $\mathbf{T}(\boldsymbol{s_n}, \boldsymbol{s_{n+1}})$.

2.1 The SAT problem

Let *Bool* denote the *boolean* domain $\{0, 1\}$, and $\mathit{Vars} := \{x_0, x_1, x_2, \ldots\}$ be a finite set of boolean variables. A *literal* is a boolean variable x_i or a negated boolean variable $\overline{x_i}$. A *clause* is a set of literals, implicitly disjoined. A *SAT instance* is a set of clauses, implicitly conjoined. A *valuation* is a function $\mathit{Vars} \to \mathit{Bool}$. A literal x_i is said to be satisfied by a valuation if its variable is mapped to 1; a literal $\overline{x_i}$ if its variable is mapped to 0. A clause is said to be satisfied if at least one of its literals is satisfied. A *model* (satisfying assignment) for a SAT instance is a valuation where all clauses are satisfied. The *SAT problem* is to find a model for a given set of clauses.

2.2 Converting formulas to SAT

There are several ways of translating a propositional formula into clauses, in such a way that satisfiability is preserved. This is typically done by introducing auxiliary variables giving names to some or all subformulas, then generating clauses that establish a definitional relation between the introduced variables and the truth-values of their respective subformulas. Any model for

the translated problem (which contains more variables) has the property that its restriction to the original set of variables yields a model for the original formula. We assume the existence of such a translation technique and introduce the following notation:

> **Definition.** By $[\varphi]^p$ we denote a set of clauses defining φ such that p is the literal representing the truth-value of the whole formula. We call p the *definition literal* of φ. Further, we write $[\varphi]$ as a short hand for $[\varphi]^p \cup \{p\}$.

For example $[x \wedge y]^p$ may be translated into the clauses $\{ \{\overline{p}, x\}, \{\overline{p}, y\}, \{p, \overline{x}, \overline{y}\} \}$.

2.3 Temporal Induction

This section briefly summarizes the verification technique *temporal induction* presented in [SSS00].[1] The word "temporal" suggests that the induction is carried out over the time steps of the FSM. Like a standard induction proof, a temporal induction proof consists of two parts: the base-case and the induction-step. In its simplest form, the base-case states that the property should hold in the initial states; and the induction-step states that the property should be preserved by the transitions of the FSM. Expressing the two parts of the induction proof as SAT-problems is straight-forward—still, the resulting method is already an interesting complement to BDD-based verification methods, especially for systems where the transition relation has no succinct BDD-representation. However, the method is not complete, since the induction-step might not be provable even though the property is true.

To make the method complete, the induction-step is strengthened in two ways. Firstly, the property is assumed to hold for a path of n successive states, rather than just one. This means that a longer base-case must be proven. Secondly, the states of the path are assumed to be unique. It follows immediately from finiteness that the second strengthening makes the method complete in the sense that there is always a length for which the induction-step is provable. Soundness is treated in detail in section 4. Let us formalize the strengthened induction by defining the following formulas:

$$\begin{aligned}
\mathbf{Base}_n &:= \mathbf{I}_0 \wedge \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_{n-1} \wedge \mathbf{T}_{n-1})\Big) \wedge \overline{\mathbf{P}_n} \\
\mathbf{Step}_n &:= \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_n \wedge \mathbf{T}_n)\Big) \wedge \overline{\mathbf{P}_{n+1}} \\
\mathbf{Unique}_n &:= \bigwedge_{i \le j \le n} (\boldsymbol{s_i} \neq \boldsymbol{s_{j+1}}) \quad = \bigwedge_{i \le j \le n} \bigvee_k \neg(s_{i,k} \leftrightarrow s_{j,k})
\end{aligned}$$

An interpretation of these formulas is depicted in *Fig. 1.* Note that when proving correctness we show that the formulas are *unsatisfiable.* In the base-case we assume that all shorter base-cases have been proved already, and add

[1] The authors use only the word "induction" in this presentation, but have later adopted the term "temporal induction" and used it in other contexts.

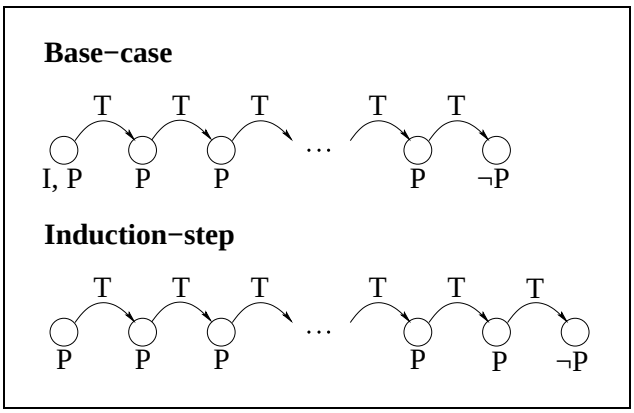

Fig. 1. If the n-th *base-case* is unsatisfiable, it should be read as "There exists no n-step path to a state violating the property, assuming the property holds the first $n-1$ steps." If the n-th *induction-step* is unsatisfiable, it should be read as "Following an n-step trace where the property holds, there exists no next state where it fails".

the property to each state as this tends to make the resulting SAT-problem easier. With these definitions, we can now state an algorithm that intertwines looking for bugs of longer and longer lengths, and trying to prove the property by deeper and deeper induction-steps:

Algorithm 1. "Temporal Induction".

for n $\in 0..\infty$ **do**
  if $(satisfiable([\mathbf{Base}_n]))$
    return PROPERTY FAILS
  if $(\neg satisfiable([\mathbf{Step}_n] \cup [\mathbf{Unique}_n]))$
    return PROPERTY HOLDS

Variations of this algorithm are also meaningful. For instance, checking only the base-case gives a pure bug-hunting algorithm, which delivers counter-examples more quickly. By altering the formula of the base-case slightly, it is possible to start at a higher n and taking bigger leaps than 1. Checking every size of n may be unnecessarily costly. If the bug or proof is deep, taking bigger leaps means solving fewer SAT-problems. However, if there is a bug, *Algorithm 1* (as stated) will always find a shortest counter-example. This may be important. In the remainder of the article, we will show how the cost of incrementing n by only 1 can be greatly reduced by solving the SAT-problems incrementally.

3 Incremental SAT

A typical stand-alone SAT-solver accepts a problem instance as input, solves it, and outputs a model or an "Unsatisfiable" statement as result. This can be inadequate if you wish to solve many similar SAT-instances. The most obvious overhead is re-parsing the (almost) same clause set over and over again. But more importantly, the same, often expensive, inferences may be carried out over and over again. Equipping the SAT-solver with an interface that

allows the next SAT-instance to be specified incrementally from the current (solved) instance will certainly remove the parsing problem, but may reduce the number of inferences too.

We focus on the type of solver introduced by [MS99], based on *conflict analysis* and *clause recording.*[2] Such a solver implements a DPLL-style backtracking search procedure [DLL62]. The idea behind augmenting the basic procedure with conflict analysis is that for every conflict detected during the search, some effort is spent on finding a *reason* for the conflict that can be encoded as a clause and added to the clause set. The *recorded* clauses will serve as a cache for the same type of conflicts in later parts of the search-space. For example, if assuming x and y to be true led to a conflict, the clause $\{\overline{x}, \overline{y}\}$ may be recorded. Assuming either x or y to be true in some later part of the search-tree, will immediately give the implied value to the other variable, avoiding repetition of the possibly lengthy derivation. The effectiveness of this idea has been empirically established by many authors. A motivation for incremental SAT is that the recorded clauses may not only be useful in later parts of the search-tree of the *same* SAT-instance, but also in a later *similar* SAT-instance.

To describe the different design issues encountered when implementing an incremental SAT-system, we adopt an object-oriented view, using a *solver object* which stores the *problem clauses* (the current SAT-instance) as well as the *learnt clauses* (the recorded clauses). The solver has methods for modifying and solving the current SAT-instance. The simplest imaginable interface would contain the following methods:

addClause	(**Clause** c)	*– will add a clause to the clause database.*
solve		*– will solve the current instance.*

Using this interface, the user is allowed to add clauses until he has specified the first SAT-problem. He can then use *solve* to check if the problem is satisfiable or not. If it is, he may add more clauses to constrain the problem further and re-run *solve*. This procedure can be repeated until all SAT instances of interest have been solved. Typically the last instance is unsatisfiable, from which point no extension can be satisfiable.

This approach to incremental SAT, introduced in [Hok93], is limited as the user can never remove anything added. Many interesting incremental SAT-problems requires some form of clause removal. Therefore [WKS01] suggested the following interface to the solver:

addClause	(**Clause** c)	
removeClause	(**Clause** c)	*– will remove an existing clause from the*
solve		*clause database.*

[2] This includes SAT-solvers such as: Grasp, Sato, zChaff, Limmat, Berkmin, and the authors' own solvers Satnik and Satzoo.

By this interface, any set of related problems can be solved incrementally. However, the ability to remove clauses clashes with conflict clause recording. The conflict analysis is guaranteed to produce clauses that are implied by the problem clause set; thus adding these clauses can never cause unsoundness. But removing problem clauses may suddenly render recorded clauses invalid. A detailed dependency analysis must therefore be carried out to remove the invalid clauses, which in turn may require extra book-keeping during the actual solving process. For a longer treatment of this approach see [WKS01].

In contrast, we propose the following interface which only enables the removal of unit clauses. The motivation is that it is *very* simple to implement (5 lines of code in our solver), while being expressive enough to encompass several interesting incremental SAT-problems not expressible by the original interface:

addClause	(**Clause** c)
solve	(**list**⟨**Literal**⟩ assumptions)

The extra list of literals passed to *solve* should be viewed as unit clauses to be added during this particular solving, then removed upon return from the solver. The reason that this approach is simpler is that *all* learned clauses are safe to keep, and thus no extra book-keeping is needed. To see why it is safe, note that the extra unit clauses can be seen (and implemented) as internal assumptions by the search procedure, and that it is an inherent property of conflict clauses that they are independent of the assumptions under which they occur. [3]

4 Incremental Induction

In section 2.3 we saw a straight-forward algorithm for proving or disproving safety properties by induction. We break this algorithm into two parts, the *base-case* ("bug-finder") and the *induction-step* ("upper-bound prover"), and show how they can be implemented incrementally using the SAT-interface of section 3.

[3] In fact, the more general interface can be simulated to a large extent. By inserting the clause $\{x\} \cup C$, and passing $\overline{x}$ as an assumption literal, we achieve the same effect as inserting C. Asserting x to be true afterwards will make the clause true forever, and it will be removed from the clause database by the top-level simplification procedure of the solver.

Algorithm 2 "Extending base".

```
addClauses([I_0])
for n ∈ 0..∞ do
    addClauses([P_n]^{p_n})
    solve({¬p_n})
    if (Satisfiable)
        return property fails
    addClause({p_n})
    addClauses([T_n])
```

Algorithm 3 "Extending step".

```
addClauses([¬P_0])
for n ∈ -1..−∞ do
    solve({})
    if (Unsatisfiable)
        return ind. step holds
    addClauses([T_n])
    addClauses([P_n])
    for i ∈ 0..n+1 do
        addClauses([s_i ≠ s_n])
```

A first observation on these algorithms is that they build the trace of states related by the transition relation in different directions (n is decremented in the step). Growing the trace forwards in the base-case allows us to keep the often strong formula $\mathbf{I}_0$ fixed in the SAT-solver. Building the trace in the opposite direction would force us to put the initial state constraints as an assumption literal to "*solve*", which will have the undesirable effect of making any recorded conflict clause depending on the initial state ineffective in successive iterations. Similarly in the step, growing the trace backwards makes it unnecessary to use any assumption literal at all, which again promotes reuse of recorded clauses between iterations.

Different top-level strategies for how to combine the two algorithms to a safety-checking procedure are possible. To emulate *Algorithm 1* of section 2.3, the algorithms could be run in parallel, each with its own solver instance. As soon as the induction-step succeeds for a particular length, an unsatisfiable base-case of that length will constitute a proof of the safety property. However, it is also possible to mix the two algorithms into one. We will then have to break the natural direction of building the trace for either the base-case or the induction-step. We arbitrarily chose to sacrifice the induction-step.

Algorithm 4 "Zig-zag".

```
addClauses([I_0]^z)          – z is the definition literal for I_0
for n ∈ 0..∞ do
    addClauses([P_n]^{p_n})  – p_n is the definition literal for P_n
    solve({¬p_n})            – step: do not include I_0
    if (Unsatisfiable)       – P_n must hold!
        return property holds
    solve({z, ¬p_n})         – base-case: include I_0
    if (Satisfiable)         – counter-example found!
        return property fails
    addClause({p_n})         – assert P_n from now on
    addClauses([T_n])        – assert transition from s_n to s_{n+1}
    for i ∈ 0..n-1 do        – add uniqueness constraints
        addClauses([s_i ≠ s_n])
```

The reason for stating this algorithm is partly to show that there is many possible ways of encoding the safety-checking procedure incrementally. With

this algorithm, the SAT-solver is allowed to share conflict clauses between the base-case and the induction-step, which may be beneficial. We include the algorithm in our benchmark section.

4.1 Discussion

We will now try to draw a map over possible induction based safety-checking algorithms. Let us use the term *bad state* for a state were the safety property does not hold. It is generally observed that checking safety properties is symmetric with respect to the initial states and the bad states. Everything presented up to this point could have been carried out backwards, with the roles of initial states and bad states exchanged, and the transition relation inverted. We are going to adopt this symmetrical view from now on.

In this view, we regard the induction-step as a method of finding an upper bound on the length of a shortest counter-example, and the base-case as a way of producing the counter-example. Now, what must a shortest counter-example look like? It has to start in an initial state, it has to end up in a bad state, and the states in between must not be either initial or bad (otherwise it could not be a shortest counter-example). Using $\mathbf{B}$ (bad) for $\overline{\mathbf{P}}$ we can view the set of possible shortest counter-examples pictorially:

$$
\begin{array}{ll}
\textit{length 0:} & \mathbf{I}\mathbf{B} \\
\textit{length 1:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{length 2:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{length 3:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
 & \ldots \\
\textit{length n:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B}
\end{array}
$$

Each line depicting a (shortest) counter-example corresponds to a conjunction of constraints $(\mathbf{I}_0 \wedge \mathbf{T}_0 \wedge \overline{\mathbf{B}_1} \wedge \overline{\mathbf{I}_1} \wedge \mathbf{T}_1 \wedge \ldots)$. There is a lot of sharing between the counter-examples of different lengths, and indeed if we remove either the initial $\mathbf{I}$ or the final $\mathbf{B}$ from the n-th counter example, i.e.:

$$
\begin{array}{lll}
 & (1) & \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{or} & (2) & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}
\end{array}
$$

then any counter-example of length n or *longer* will include all the constraints of (1) and (2). This means that if either the constraints of (1) or (2), or any *subset* of these, yields an unsatisfiable problem, then so will *all* possible shortest counter-examples of longer lengths. Thus we have found an upper bound on the shortest counter-example.

The picture above does not contain all constraints derivable from the fact

that we are considering a *shortest* counter-example. We can further conclude:

1. Between no two states is there a shorter path.

or weaker 2. Between no two non-neighbors is there a transition (and the last state is unique).

or weaker 3. No two states are the same.

Any of these facts can be used when proving an upper bound. As long as we keep adding constraints that must be fulfilled by shortest counter-examples, any contradiction reached means we have established an upper bound. The reason for stating weaker versions of the shortest-path requirement is that these versions can be implemented more efficiently. Furthermore, we have already noted that the third condition is enough to make the procedure complete. In the next section we describe how the implementation of this condition can be improved.

Taking this subset-of-counter-example view, the induction-step we have used in our algorithms can now be viewed as selecting the subset of (1) not containing any $\overline{\mathbf{I}}$:s but including the uniqueness constraints dictated by condition 3.[4] Through experiments we found that this choice worked well in practice.

4.2 Finding a counter-example

If the user knows or has reason to believe that the property is false, he may want to run just the base-case to quickly produce a counter-example. In this case, it is less clear if any extra constraints should be added to the trace. In *Algorithm 1* and *2* we chose to add $\mathbf{P}$. More constraints mean more clauses in the solver, which leads to slower propagation, but also to a smaller search-tree. Which of the two effects is predominant in a particular case is hard to judge. In general, adding weak constraints is seldom a good idea.

Present BMC tools can optionally produce a SAT-problem stating that the property fails among the first n steps rather than after exactly n steps. Care must be taken before adding extra constraints to such formulations. For instance, one can no longer require the states to be unique. One must also assume (or modify) the transition relation to always have a next state; or risk getting an unsatisfiable problem due to deadlock, even in the presence of a bug. A comparison between this "one-shot" method and the incremental base-case is included in our experiments.

4.3 Improving the Unique States Requirement

The uniqueness constraints described in section 2.3 and used in *Algorithm 1, 3* and *4* require each pair of states to be different. These requirements are

[4] The *recurrence diameter* introduced in [BCCZ99] can similarly be viewed as the subset containing only the $\mathbf{T}$:s together with uniqueness constraints.

statically added, and their number will grow quadratically in the length of the induction-step. For problems requiring high induction length, there is a risk of adding numerous possibly superfluous constraints that will tax the SAT-solver heavily. We propose a *dynamic* approach where the models returned by the solver in the induction-step are examined, and only if two states are actually equal, a constraint stating that they should be different is added. The solver must then be run again, which may possibly cost more than adding superfluous constraints, but hopefully the incrementality of the approach means that any re-run is very quick. We verified experimentally that the method indeed seems to perform better in general.

A question that has not been treated sufficiently in earlier presentations on induction is what variables should be included in the uniqueness constraints. It is not unusual to describe the FSM in the form of a sequential circuit. The standard interpretation of a circuit is to consider both the latches (the state holding elements) and the inputs as *state variables* of the FSM. However, it is fairly clear that there is no need to include inputs in the uniqueness constraints. If two states are equal except for the inputs, whatever value the inputs assume in the second state, they could have assumed in the first. It is therefore safe to require only the latch-variables do be different—a much stronger condition. In fact, this is often what is implemented [CS00]. Note that failing to remove the superfluous state variables from the uniqueness constraints gives an ineffective induction algorithm, as each extra state variable has the potential of doubling the depth needed to prove the step.

If on the other hand the FSM is given as two propositional formulas $\mathbf{I}$ and $\mathbf{T}$ it is less clear what variables can be excluded.[5] We propose the following solution:

1. Include only variables occurring *both* in the current and the next state of the transition relation.
2. Do not add uniqueness constraints including the first or the last state of the trace.

We refer to uniqueness constraints over this reduced set of state variables as *strong uniqueness*.

4.4 Correctness

We will now prove that temporal induction with strong uniqueness is sound. Recall that the induction-step can be strengthened by anything that holds for a shortest counter-example. It then suffices to show that a counter-example that is not strongly unique cannot be shortest. Let us introduce the following notation:

[5] The result of parsing an SMV file often leaves you with just this.

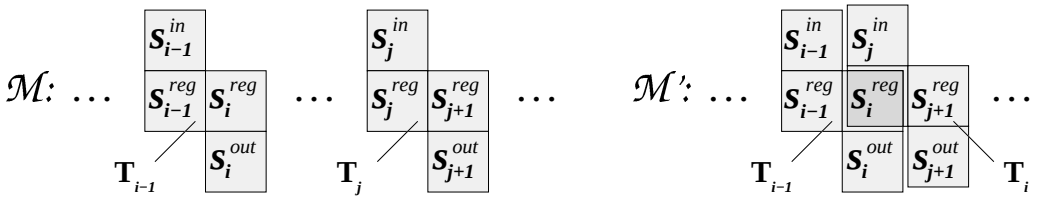

Fig. 2. The picture shows the contraction of the counter-example $\mathcal{M}$ to $\mathcal{M}'$. The state variables constrained by the transition relations at the point of "gluing" are printed in the boxes; the remaining trace is represented by the "...".

$$
\begin{array}{llll}
\boldsymbol{s_i}^{left} & := & vars(\mathbf{T}_i) \cap \boldsymbol{s_i} & \boldsymbol{s_i}^{in} := \boldsymbol{s_i}^{left} \setminus \boldsymbol{s_i}^{right} \\
\boldsymbol{s_i}^{right} & := & vars(\mathbf{T}_{i-1}) \cap \boldsymbol{s_i} & \boldsymbol{s_i}^{out} := \boldsymbol{s_i}^{right} \setminus \boldsymbol{s_i}^{left} \\
 & & & \boldsymbol{s_i}^{reg} := \boldsymbol{s_i}^{left} \cap \boldsymbol{s_i}^{right}
\end{array}
$$

Let $\mathcal{M}$ be the model of a formula encoding a counter-example of depth n:

$$\mathcal{M} \models \mathbf{I}_0 \wedge \mathbf{T}_0 \wedge \mathbf{T}_1 \wedge \ldots \wedge \mathbf{T}_{n-1} \wedge \mathbf{B}_n.$$

We now show by construction that if $\mathcal{M} \models (\boldsymbol{s_i}^{reg} = \boldsymbol{s_j}^{reg})$ for some $0 < i < j < n$ ($\mathcal{M}$ is not strongly unique) then there is a shorter counter-example. Define $\mathcal{M}'$ over $\{\boldsymbol{s_0}, \ldots, \boldsymbol{s_{n-(j-i)}}\}$ as follows:

$$
\begin{array}{lcll}
\mathcal{M}'(\boldsymbol{s_k}) & = & \mathcal{M}(\boldsymbol{s_k}) & ,k < i \\
\mathcal{M}'(\boldsymbol{s_k}) & = & \mathcal{M}(\boldsymbol{s_{k+(j-i)}}) & ,k > i \\
\mathcal{M}'(\boldsymbol{s_i}^{in}) & = & \mathcal{M}(\boldsymbol{s_j}^{in}) & \\
\mathcal{M}'(\boldsymbol{s_i}^{out}) & = & \mathcal{M}(\boldsymbol{s_i}^{out}) & \\
\mathcal{M}'(\boldsymbol{s_i}^{reg}) & = & \mathcal{M}(\boldsymbol{s_i}^{reg}) &
\end{array}
$$

$\mathcal{M}'$ now constitutes a counter-example of depth $n-(j-i)$. We have contracted the counter-example by simply removing all states between i and j (depicted in *Fig. 2*). The only potential problem lies in the "gluing" of the head and the tail at state i. However, the only constraints containing $\boldsymbol{s_i}$ are $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$. But $\mathbf{T}_{i-1}$ does not contain any variables from $\boldsymbol{s_i}^{in}$, so letting $\mathcal{M}(\boldsymbol{s_i}^{in}) \neq \mathcal{M}'(\boldsymbol{s_i}^{in})$ cannot make $\mathbf{T}_{i-1}$ false in $\mathcal{M}'$. Similarly for $\mathbf{T}_i$ which does not contain any variables from $\boldsymbol{s_i}^{out}$. Finally $\mathcal{M}(\boldsymbol{s_i}^{reg}) = \mathcal{M}(\boldsymbol{s_j}^{reg})$, so indeed $\mathcal{M}'$ must be a model for the constraints $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$. □

The proof can easily be extended to establish that the exclusion of the first and the last state is superfluous if all variables of $\mathbf{I}$ occur in the next state of $\mathbf{T}$ and all variables of $\mathbf{B}$ occur in the current state of $\mathbf{T}$.

5 Experimental Results

The ideas presented in this paper were implemented in the prototype tool Tip [6] which was integrated with the SAT-solver Satzoo. All benchmarks were performed on a 2 GHz Pentium 4 with 512 MB of memory running Linux. We set the time-out for all launches to 10 minutes, and the memory limit to 400 MB. The benchmarks were collected from several sources. In the tables, each benchmark name is tagged with the source of the problem:

cadence	–	Example files from the Cadence SMV distribution.
cmu	–	Example files from the CMU SMV distribution.
ken	–	SMV case studies from Ken McMillan's web-page.
nusmv	–	Example files from the NuSMV distribution.
vis	–	Example files from the VIS distribution.
texas	–	The *Texas 97 benchmarks* from Berkeley University.
eijk	–	ISCAS'89 sequential equivalence checking from [Eijk98].
irst	–	Problems from the Model Checking Group at IRST.

All problems were converted to flat SMV-format with only boolean variables and no sub-modules. For each problem, the safety properties were extracted. In this process, CTL formulas "EF" were changed into "AG¬" and all fairness constraints were removed. Different properties for the same system are indicated by a subscript after the system name.

Counting each property as a separate instance, a total of 185 problem instances were collected. As our first experiment, we ran Tip, Cadence SMV, CMU SMV, and NuSMV on each of these instances. All tools were run with a default set of options, providing no problem specific variable ordering:

`Tip`	*filename*
`CadSMV`	*filename*
`CmuSMV`	`-reorder` *filename*
`NuSMV`	`-AG -dynamic -coi` *filename*

Instances solved in less than 1 second by all tools were considered trivial and removed, leaving 158 instances.

5.1 Comparison with BDD-tools

The result of the comparative experiment is presented in *Table 1.* The default strategy of Tip runs the base-case and the induction-step presented in *Algorithm 2* and *3* in parallel, each with its own solver instance. The two algorithms are given equal amount of CPU time, until the point where either the base-case fails, and a counter-example is found, or the induction-step is

[6] The tool Tip, the SAT-solver Satzoo and all benchmarks used in this article can be downloaded from *http://www.cs.chalmers.se/∼een/*

proven, and the remaining base-cases (if any) are proved with 100% CPU.

The purpose of the experiment was to relate the performance of induction to industrially applied methods, and to show the (lack of) correlation between hardness for BDD-based methods and hardness for induction-based methods. TIP was able to solve 6 instances where BDD-based verification failed, showing that induction may be a valuable complementary method. [7]

5.2 Effect of incrementality

The second experiment we performed was a comparison of *Algorithm 2* and *3* using the incremental interface of SATZOO and using SATZOO as an external solver. In this experiment, we used only problem instances where the property held. The result is presented in *Table 2*.

The experiment establishes a substantial speed-up by the incremental approach. Unsurprisingly, the gain was larger for instances where a long induction-step was needed to prove the property.

From the table we can also see that the induction-step usually takes longer to prove than the base-case. We observed the same behavior for instances where the property failed (although not presented here). This is the reason the default strategy of TIP does not increase the lengths of the step and base evenly, but instead devotes the same amount of CPU to each. Otherwise, bugs may not be found due to hard (and futile) induction-steps.

5.3 One solver instance or two

The third experiment compared *Algorithm 4* ("Zig-Zag") using one solver instance to running the induction-step and the base-case in separate solver instances. ("Dual"). In this experiment, the step and the base were incremented evenly so that both methods would solve only the minimal number of SAT-instances. We also include the standard implementation of (complete) induction as presented in [SSS00]. The results are also in *Table 2*.

The experiment suggests that separate solver instances for the base and the step is favorable. From the table we can also see that the incremental implementation of induction clearly outperforms the standard implementation.

5.4 BMC Comparison

In the fourth experiment, we compared incremental search for counter-example to the "one-shot" approach described in section 4.1. The result is presented in *Table 3*. The experiment shows that often you must know the exact length of a shortest counter-example for the one-shot method to be advantageous.

[7] These problems were all "TCAS II" problems from the NUSMV distribution, originally used in "Model Checking Large Software Specifications" [CAB98].

Tool	Solved (of 158)	Alone in solving
Cadence SMV	131	5
Tip	92	6
CMU-SMV	90	0
NuSMV	73	0

Table 1. *Tool comparison.* The left column shows the total number of solved instances within 10 minutes. The right column show how many of these instances no other tool could solve. Cadence SMV excelled by proving 22 instances that neither of the two other SMVs could prove, and 39 more instances than Tip. Still only 5 instances were unique, as Tip solved many of the problems where NuSMV and CMU-SMV failed, plus 6 that Cadence SMV did not solve.

Name	Len	Step^{inc}	Step^{ext}	Base^{inc}	Base^{ext}	Dual	ZigZag	StdInd
cmu:periodic	97	70.7	[>600]	10.7	141.8	80.9	[>600]	[>600]
eijk:S208c	259	448.0	[>600]	[>600]	[>600]	[>600]	[>600]	[>600]
eijk:S208o	258	483.2	[>600]	[>600]	[>600]	[>600]	564.2	[>600]
eijk:S208	259	436.7	[>600]	[>600]	[>600]	[>600]	503.7	[>600]
eijk:S298	59	27.7	[>600]	34.9	96.2	62.9	316.1	[>600]
eijk:S510	11	5.2	8.0	0.5	0.9	5.9	7.4	10.1
eijk:S820	12	6.1	22.9	6.4	12.5	12.6	20.2	30.1
eijk:S832	12	7.6	28.2	5.8	12.9	13.4	25.1	35.2
eijk:S953	8	1.7	4.2	0.1	0.2	1.9	4.2	4.4
ken:oop$_1$	30	39.4	[>600]	0.3	7.4	39.9	492.0	254.0
nusmv:guidance$_1$	11	2.8	10.2	0.8	3.4	3.5	3.9	11.1
nusmv:guidance$_7$	28	120.3	[>600]	315.0	[>600]	438.9	[>600]	[>600]
nusmv:tcas$_2$	7	1.3	3.1	0.2	0.3	1.5	1.9	4.3
nusmv:tcas$_3$	6	1.3	3.3	0.0	0.1	1.3	1.8	3.2
texas:parsesys$_2$	4	12.2	13.5	0.2	0.2	14.7	12.5	7.8
vis:prodcell$_{12}$	30	256.6	[>600]	112.8	445.5	367.3	[>600]	[>600]
vis:prodcell$_{13}$	9	4.6	12.4	0.1	0.6	4.8	3.7	14.7
vis:prodcell$_{14}$	17	31.3	185.1	7.3	14.2	38.7	52.3	219.9
vis:prodcell$_{15}$	24	109.3	[>600]	23.0	80.1	132.4	216.7	[>600]
vis:prodcell$_{16}$	6	2.1	4.1	0.0	0.1	2.1	1.2	4.7
vis:prodcell$_{17}$	28	211.3	[>600]	52.4	277.5	265.0	[>600]	[>600]
vis:prodcell$_{18}$	14	21.4	117.9	0.4	3.2	21.8	28.6	128.9
vis:prodcell$_{19}$	23	61.6	457.0	23.4	86.0	85.0	178.5	[>600]
vis:prodcell$_{24}$	38	391.9	[>600]	[>600]	[>600]	[>600]	[>600]	[>600]

Table 2. *Experimental results for the effect of incremental SAT vs. external SAT.* All times are in seconds. The experiment includes all instances where the property was proved to hold in in the first experiment. Launches where all methods took less than 3 seconds have been left out. "Dual" stands for running one iteration of *Alg.2* and *Alg.3* interchangeably; "ZigZag" refers to *Alg.4*; "StdInd" stands for standard induction with all uniqueness constraints statically added and using an external SAT-solver.

Name	Length	Incremental BMC	Perfect Guess	25%-off Guess
nusmv:tcas$_1$	11	3.6	3.7	5.0
nusmv:tcas$_4$	15	9.7	9.7	18.2
nusmv:tcas$_5$	24	48.7	40.1	125.2
nusmv:tcas$_6$	17	13.6	13.5	38.2
texas:parsesys$_1$	10	9.3	0.8	1.1
texas:parsesys$_3$	9	3.3	0.7	0.9
texas:two-proc$_2$	16	4.7	1.0	2.9
texas:two-proc$_4$	20	20.9	1.8	9.1
vis:eisenberg	20	20.7	18.1	79.1

Table 3. *Experimental result for incremental BMC vs. SAT-instances of fixed length.* All times are in seconds. "Perfect Guess" means the SAT-instance encode "there is a bug of length $\leq k$" where k is the length of the shortest counter-example. "25%-off" means k is multiplied by 1.25. Launches where all methods took less than 3 seconds have been left out.

Name	Len	Time^d	Time^s	Ban^d	Ban^s	Clau^d	Clau^s	Conf^d	Conf^s
cmu:periodic	97	70.7	120.4	0	4656	455k	908k	15k	14k
eijk:S208	259	436.7	[>600]	258	[>20000]	186k	-	76k	-
eijk:S298	59	27.7	66.6	114	1653	69k	296k	24k	25k
ken:oop$_1$	30	39.4	50.4	113	406	67k	101k	32k	30k
nusmv:guidance$_7$	28	120.3	66.9	0	378	151k	276k	56k	28k
vis:prodcell$_{12}$	30	256.6	252.7	0	406	346k	439k	48k	43k
vis:prodcell$_{14}$	17	31.3	41.7	0	120	189k	217k	11k	13k
vis:prodcell$_{15}$	24	109.3	134.3	0	253	273k	330k	29k	29k
vis:prodcell$_{17}$	28	211.3	253.6	0	351	322k	400k	45k	46k
vis:prodcell$_{18}$	14	21.4	25.5	0	78	153k	171k	10k	10k
vis:prodcell$_{19}$	23	61.6	71.9	0	231	260k	311k	18k	18k
vis:prodcell$_{24}$	38	391.9	490.1	0	666	440k	588k	60k	61k

Table 4. *Experimental results for dynamic vs. static uniqueness constraints in the induction-step.* All times are in seconds. Launches taking less than 10 seconds or having shorter length than 5 has been left out. A superscript "d" means dynamic (on demand) adding of uniqueness constraints. A superscript "s" means static adding of uniqueness constraints between all pairs of states. "Ban" is the number of constraints added (banning two states from being equal). "Clau" is the final number of clauses in the solver. "Conf" is the total number of conflicts in the search-tree of the solver. Only three problems actually needed uniqueness constraints to be provable, and in almost all other cases it incurred a cost to add them. For the three cases where the constraints were necessary, adding them dynamically lead to a speed-up. Without uniqueness constraints these three problem are not provable by induction. The dynamic method thus saves the user from guessing for each problem if uniqueness constraints should be used or not without incurring any extra cost.

5.5 Uniqueness constraints

In the final experiment, we studied the effect of adding uniqueness constraints dynamically and statically, including both instances where the constraints must be added, and instances which are provable without uniqueness constraints. The result is presented in *Table 4*.

The effect of sharpening the constraints by removing variables are not presented, as it is clearly advantageous. A study of the "*eijk*" equivalence checking problems, where 9 out of 13 need uniqueness constraints, showed that *none* of these could be solved within the time-bound without using the sharpening.

6 Related Work

Incremental BMC was independently introduced by Ofer Strichman in [Stri01] and Sakallah et. al. in [WKS01]. Our approach differs from previous attempts in that we keep all clauses from previous iterations (including conflict clauses). Moreover, we complete the method with incremental temporal induction. Strichman's work further includes several techniques to enhance the SAT-solving of BMC problems, including *internal constraints replication* for copying invariant conflict clauses between the time steps of the trace, and BMC specific variable decision strategies [Stri00].

7 Conclusions

Temporal induction has been used before to prove upper bounds for BMC [SSS00]. In these efforts, the authors established it too costly to gradually increase the depth of the induction proof using an external SAT-solver. We have shown that integrating the SAT-solver and the induction procedure overcomes this cost. Furthermore, we sharpened the unique-states constraints by a syntactic analysis on the transition relation; an improvement that was absolutely necessary for many of our benchmarks to go through.

By extensive testing we further reinforced the view that induction is an important complement to BDD-based methods for safety-checking. The combination of techniques presented in this paper results in what the authors believe to be the first efficient and complete induction based checker produced by academia. Enabled by the incremental SAT-interface, we explored an online method of adding uniqueness constraints on demand. To a large extent the method saves the user from deciding manually whether or not to add these constraints, making temporal induction a more push-button technique.

As a side-effect of implementing temporal induction incrementally, we got an incremental BMC for safety properties. The efforts on incremental BMC by [Stri01,WKS01] was based on extensive adaptation of the underlaying SAT-solver. We have shown that results of the same magnitude can be achieved by a much smaller modification of the solver. A standard way of applying BMC

is to generate a single SAT-problem encoding the presence of a bug within k time steps. We have compared this method to iterating up to k incrementally and found that the incremental approach was faster in most cases, even if k was specified as close as 25% above the length of a shortest counter-example.

8 Future Work

The single most significant factor for the success of temporal induction is the induction depth needed. We therefore believe the most important direction of research is towards methods of automatically strengthening the induction-step in order to reduce this depth. A successful method achieving this was presented in [Eijk98,BC00]. It works by finding invariant equivalences or implications between the state variables and internal points. Casting this method into our incremental system looks very promising. Stronger constraints on the shape of a shortest counter-example were suggested in [SSS00], but have not yet been successfully applied. We would like to investigate if a dynamic approach similar to that we used for uniqueness constraints might be helpful.

Finally, there are many possible ways of tuning the SAT-solver to incremental temporal induction. In particular, we wish to explore native uniqueness constraints, as well as the methods presented in [Stri00,Stri01] for specialized variable orderings and constraint replication.

Acknowledgments

We would like to thank Per Bjesse and Mary Sheeran for their careful reading and valuable criticism of the manuscript for this paper.

References

[BC00] P. Bjesse, K. Claessen. **"SAT-based Verification without State Space Traversal"** in *Formal Methods in Computer-Aided Design*, LNCS:1954, Springer-Verlag 2000.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. **"Symbolic model checking without BDDs"** in *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS:1579, Springer-Verlag 1999.

[BCRZ99] A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu. **"Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs"** in *Proc. 11th Int. Conf. on Computer Aided Verification*, LNCS:1633, Springer-Verlag 1999.

[Bry86] R.E. Bryant. **"Graph-based algorithms for boolean function manipulation"** in *IEEE Trans. on Computers*, C-35(8), Aug. 1986.

[CAB98] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J.D. Reese **"Model Checking Large Software Specifications"** in *IEEE Tran. on Software Engineering* 24(7), Jul. 1998

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y. Vardi. **"Benefits of bounded model checking at an industrial setting"** in *Proc. 13th Int. Conference on Computer Aided Verification*, LNCS:2102, 2001.

[CS00] K. Claessen, M. Sheeran. **"A Tutorial on Lava: A Hardware Description and Verification System"** at *http://www.cs.chalmers .se/~koen/Lava*, 2000

[DLL62] M. Davis, M. Logman, D. Loveland. **"A machine program for theorem proving"** in *Communications of the ACM*, vol 5, 1962.

[Eijk98] C.A.J. van Eijk. **"Sequential equivalence checking without state space traversal"** in *Proc. Conf. on Design, Automation and Test in Europe*, 1998.

[Hok93] J.N. Hooker **"Solving the Incremental Satisfiability Problem"** in *Journal of Logic Programming*, vol 15, 1993.

[MS99] J.P. Marques-Silva, K.A. Sakallah. **"GRASP: A Search Algorithm for Propositional Satisfiability"** in *IEEE Transactions on Computers*, vol 48, 1999.

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik **"Chaff: Engineering an Efficient SAT Solver"** in *Proc. of the 38th Design Automation Conference*, 2001.

[Stri00] O. Strichman **"Tuning SAT checkers for Bounded Model Checking"** in *Proc. of 12th Intl. Conf. on Computer Aided Verification*, LNCS:1855, Springer-Verlag 2000

[Stri01] O. Strichman **"Pruning techniques for the SAT-based Bounded Model Checking Problem"** in *Proc. 11th Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, 2001.

[SSS00] M. Sheeran, S. Singh, G. Stålmarck. **"Checking safety properties using induction and a SAT-solver"** in *Formal Methods in Computer Aided Design*, LNCS:1954, Springer-Verlag 2000.

[WKS01] J. Whittemore, J. Kim, K. Sakallah. **"SATIRE: A New Incremental Satisfiability Engine"** in *Proc. 38th Conf. on Design Automation*, ACM Press 2001.

BMC via On-the-Fly Determinization

Toni Jussila [a,1,2] Keijo Heljanko [b,2,3] Ilkka Niemelä [a,2]

[a] *Helsinki University of Technology*
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland

[b] *University of Stuttgart*
Institute for Formal Methods in Computer Science
Universitätsstr. 38, D-70569 Stuttgart, Germany

Abstract

The paper develops novel bounded model checking techniques for labelled transition systems. The aim is to increase the efficiency of BMC by exploiting the inherent concurrency in the product of LTSs in order to cover more executions of the product within a given bound. This is done by considering a non-standard execution model, step executions, where multiple actions can take place simultaneously and where component LTSs are determinized on-the-fly, i.e., a component may be in a set of states in a step instead of in just one as in standard interleaving executions. Step executions can be further restricted to a subclass called process executions without loosing reachable states. For bounded model checking of reachability properties of the product of LTSs the paper presents translation schemes from LTSs to a constrained Boolean circuit such that satisfying valuations of the circuit correspond to step (process) executions of the product. The translation schemes have been implemented and some experimental comparisons performed. The results show that the bound needed for step and process executions is in most cases lower than in interleaving executions and that the running time of the model checker using process executions is smaller than using steps. Moreover, the performance compares favorably to a state-of-the-art interleaving BMC implementation in the NuSMV system.

[1] The financial support from Nokia Foundation and Helsinki Graduate School of Computer Science and Engineering is gratefully acknowledged.
[2] The financial support from the Academy of Finland (Project 53695) is gratefully acknowledged
[3] The financial support from the Academy of Finland (grant for research work abroad) is gratefully acknowledged

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) is a verification technique that considers only executions of bounded length of the chosen formalism [1]. The general model checking problem for linear temporal logic (LTL) is known to be **PSPACE**-complete, but the bounded case is in **NP** (assuming the used bound to be given in unary encoding). The very idea is to compile the system under verification, the property to be verified and a bound k on the length of the execution to a propositional formula having a model iff the system has an execution of length k that violates the property. The methodology has been successfully applied in industrial setting [2,3].

The aim of the paper is to develop efficient BMC techniques for systems modeled as products of labeled transition systems (LTSs) by exploiting the inherent concurrency in the systems. The basic idea is to cover more executions of a system within a given bound in a way that the size of the encoding is not substantially increased, i.e., it remains linear w.r.t. the bound. The standard approach to BMC is to use interleaving executions where exactly one action is occurring at a time. Here the idea is to encode interleaving executions more compactly by allowing multiple occurrences of actions in different components of the system simultaneously. This kind of an approach has already been investigated using 1-safe Petri nets as the system model and employing step and process executions of Petri nets with encouraging results [10,9].

The novelty in this paper is a technique that exploits independence of actions in the synchronizing product of LTSs so that multiple independent actions can take place in different component LTSs simultaneously. This technique is further combined with an on-the-fly determinization construction where for each component a set of states in which that component can be is maintained. By using determinization the number of different executions the product can have is potentially dramatically reduced, and furthermore invisible transitions do not contribute to the length of an execution. In this work, the concurrent executions of independent actions combined with on-the-fly determinization of components are called *step executions*. Without compromising reachable states, step executions can be further restricted to *process executions* satisfying an extra condition on visible actions taking place simultaneously.

Based on these ideas a technique for bounded model checking of reachability properties of the synchronizing product of LTSs is developed by devising a translation scheme from the LTSs to a constrained Boolean circuit [12] such that satisfying valuations of the circuit correspond to step executions of the product. A minor extension of the mapping handles process executions. In both cases the size of the encoding is linear w.r.t. the bound. For the encoding, Boolean circuits are employed for clarity and compactness. Such circuits can be translated to propositional formulae in CNF with a linear blow-up by introducing additional propositional variables using standard techniques [12].

The approach has been applied to a set of examples and the data obtained justify the following points. Firstly, the bound needed for step and process executions is in most cases lower than in the traditional interleaving model. Secondly, the running times using process executions are often smaller than using steps. Finally, the results compare favorably to the running times of a state-of-the-art interleaving BMC implementation [6].

The paper is organized as follows. Section 2 introduces the formalism used as the modeling language and Sect. 3 Boolean circuits. Section 4 presents the encoding schemes for both execution models. Section 5 gives test results comparing step and process executions to NuSMV [5,6] and finally Sect. 6 concludes.

2 System Modeling Formalism

Concurrent systems specified as labelled transition systems (LTS) are studied in this paper. Three execution models for the synchronized product of n LTSs are introduced. The first is the standard interleaving semantics. Thereafter, the step and process models allowing independent actions to take place simultaneously are defined. The section ends with an analysis on the relation between the different models.

Definition 2.1 An LTS is a 4-tuple $L = (S, I, \Gamma, \Delta)$ where

- S is a non-empty set of states,
- $I \subseteq S$ is a non-empty set of *initial* states,
- Γ is a non-empty set of visible actions, and
- $\Delta \subseteq S \times (\Gamma \cup \{\tau\}) \times S$, is the *transition relation*, the elements of which are called transitions of L, where τ is the invisible action.

Given n LTSs $L_1, L_2, \ldots, L_n$, $(L_1 \| \ldots \| L_n)$ is used to denote their *synchronized product* defined in the standard way, see e.g. [4] where the states of the product are n-tuples of the states of the components and where a visible action can occur iff all the components containing that action participate. However, in this work the interest is in the finite executions of the product. Firstly, the standard model of interleaving executions are defined.

Definition 2.2 Let $L = (L_1 \| \ldots \| L_n)$ be the synchronized product of n LTSs. A (finite) interleaving execution σ_{I} from a state s_1 to a state $s_{(k+1)}$ of L is a sequence

$$s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_k} s_{(k+1)} \tag{1}$$

such that each $s_i = (s_i^1, \ldots, s_i^n), s_i^j \in S_j$, i.e., each s_i^j is a state of LTS L_j and $a_i \in \Gamma_1 \cup \cdots \cup \Gamma_n \cup \{\tau\}$. In addition

- for each LTS $L_j, s_1^j \in I_j$,

- for all actions a_i and LTS L_j, if $a_i \in \Gamma_j$ then there is a transition $(s_i^j, a_i, s_{(i+1)}^j) \in \Delta_j$, otherwise, $s_{(i+1)}^j = s_i^j$, and
- for all actions a_i, if $a_i = \tau$ then there is an LTS L_j such that $(s_i^j, \tau, s_{(i+1)}^j) \in L_j$ and for any other LTS $L_l, l \neq j, s_{(i+1)}^l = s_i^l$.

Let $pr(\sigma_{\mathrm{I}})$ denote the concatenation of the visible actions in σ_{I} in the order mandated by σ_{I}.

A state s' is said to be reachable iff s' is one of the initial states $I = I_1 \times \ldots \times I_n$ or there is an execution σ from an initial state s to s'. A state s' is a *deadlock* state iff it is reachable and there is no transition $(s', a, s'') \in \Delta$.

Definition 2.3 Let $L = (S, I, \Gamma, \Delta)$ and $S' \subseteq S$. The τ-closure of S' is the set of states $S'' \subseteq S$ such that $s \in S''$ iff $s \in S'$ or there is an execution from some state in S' to s containing only τ-transitions.

The following definition presents the step executions of the synchronized product of n LTSs. The model is such that while operating on possibly non-deterministic LTSs it determinizes them on-the-fly. Therefore, in each position in the execution each component may be in a set of states instead of just one.

Definition 2.4 Let $L = (L_1 \| \cdots \| L_n)$ be the synchronized product of n LTSs. A finite *step* execution σ_{S} of L is a sequence

$$V_1 \stackrel{A_1}{\rightarrow} V_2 \ldots V_k \stackrel{A_k}{\rightarrow} V_{(k+1)} \tag{2}$$

such that each V_i is an n-tuple $(S_i^1, \ldots, S_i^n)$, $S_i^j \subseteq S_j$, $1 \leq j \leq n$, i.e., each S_i^j is a set of states of LTS L_j and each $\emptyset \subset A_i \subseteq \Gamma_1 \cup \cdots \cup \Gamma_n$. In addition the following conditions hold:

- In V_1 every S_1^j is the τ-closure of I_j.
- For each A_i and L_j, $|A_i \cap \Gamma_j| \leq 1$, i.e., in each step at most one visible action is executed from each LTS.
- For each A_i, if $a \in A_i$, then for each L_j such that $a \in \Gamma_j$ there is a transition $(s_j, a, s_j') \in \Delta_j$ such that $s_j \in S_i^j$. Furthermore $S_{(i+1)}^j$ is the τ-closure of the set of states reached via all the transitions $(s', a, s'') \in \Delta_j$ such that $s' \in S_i^j$.
- For each A_i and L_j, if $A_i \cap \Gamma_j = \emptyset$ then $S_{(i+1)}^j = S_i^j$.

The length of σ_{S}, denoted by $|\sigma_{\mathrm{S}}|$, is k. Let $lin(\sigma_{\mathrm{S}})$ denote the set of all possible linearizations of σ_{S}, i.e., the set of strings $a_1 a_2 \ldots a_k$ such that $a_i \in lin(A_i)$, for each $i = 1, \ldots, k$ where $lin(A_i)$ is the set of strings obtained by concatenating the elements in A_i in any order.

Definition 2.5 Let $L = (L_1 \| \cdots \| L_n)$, $s = (s_1, \ldots, s_n)$ and $V = (S_1', \ldots, S_n')$, $s_j \in S_j$, $S_j' \subseteq S_j$, $1 \leq j \leq n$. Define $s \sqsubset V$ to mean that each $s_j \in S_j', 1 \leq j \leq n$.

The following theorems characterize how interleaving and step executions relate to each other. They assume $L = (L_1 \| \ldots \| L_n)$.

Theorem 2.6 *Let σ_{I} be an (interleaving) execution (1) of L and $|\sigma_{\mathrm{I}}| = k$. Then there is a step execution σ_{S}*

$$V_1 \stackrel{\{a_1\}}{\rightarrow} V_2 \ldots V_l \stackrel{\{a_l\}}{\rightarrow} V_{(l+1)} \tag{3}$$

of L such that $a_1 a_2 \ldots a_l = pr(\sigma_{\mathrm{I}})$, $l \leq k$ and $s_{(k+1)} \sqsubset V_{(l+1)}$.

Theorem 2.7 *Let σ_{S} be a step execution of L reaching $V_{(k+1)}$. Then for every state $s \sqsubset V_{(k+1)}$ there is an interleaving execution σ_{I} of L reaching s such that $pr(\sigma_{\mathrm{I}}) \in lin(\sigma_{\mathrm{S}})$.*

Corollary 2.8 *A state s of $L = (L_1 \| \ldots \| L_n)$ is reachable iff there is a step execution $V_1 \stackrel{A_1}{\rightarrow} V_2 \stackrel{A_2}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{(k+1)}$ such that $s \sqsubset V_{(k+1)}$ for some k.*

The set of step executions for a system contains in most cases different elements intuitively corresponding to the same concurrent behavior. The following addition to Definition 2.4 limits the size without compromising reachable states.

Definition 2.9 A *process* execution of L is a step execution of L fulfilling the following condition

- Whenever $a_i \in A_i, i > 1$ then there is an LTS $L_j \in L$ such that $a_i \in \Gamma_j$ and there is an action $a_k \in A_{i-1} \cap \Gamma_j$.

A step execution that is not a process execution would be characterized by the fact that in some global state every LTS participating in an action a would be in a state where it could take place. It would not, though, be chosen for immediate execution, but the relevant components would remain in the same states for some steps and only then execute a.

Theorem 2.10 *Let σ_{S} be step execution of reaching state V. Then there is a process execution σ_{P} reaching V such that $|\sigma_{\mathrm{P}}| \leq |\sigma_{\mathrm{S}}|$.*

Corollary 2.11 *A state s of $L = (L_1 \| \ldots \| L_n)$ is reachable iff there is a process execution $V_1 \stackrel{A_1}{\rightarrow} V_2 \stackrel{A_2}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{(k+1)}$ such that $s \sqsubset V_{(k+1)}$ for some k.*

Intuitively the process executions are step executions which are in a certain canonical normal form. In fact, this canonical normal form corresponds exactly to the so called Foata normal form [8] from the theory of Mazurkiewicz traces, and also to a partial order semantics for 1-safe Petri nets called processes. For more on this connection, see [9] and further references there.

Fig. 1 gives two LTSs, both having the visible actions $\Gamma_1 = \Gamma_2 = \{a, b\}$. They will be used as a running example when the elements of the encoding are presented. The encoding assumes, without loss of generality that each visible transition is given a unique label. In the figure, that label is given together with the action associated with the transition.

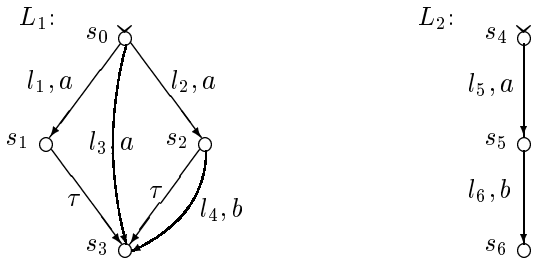

Fig. 1. Running Example

3 Boolean Circuits

The synchronized products of LTSs are translated to Boolean circuits. This section, based on the presentation of [12], introduces the concept and the associated terminology. A *Boolean circuit* is a directed acyclic graph where the nodes are called *gates*. The gates can be divided to three categories:

- *input gates* that have no incoming edges nor an associated Boolean function,
- intermediate gates that have both incoming and outgoing edges and an associated Boolean function and
- *output gates* with incoming edges and an associated Boolean function but no outgoing edges.

A *truth valuation* for a circuit with gates $\mathcal{V}$ is a function $\tau : \mathcal{V} \to \{\text{true}, \text{false}\}$. A valuation is *consistent* with the circuit if $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ for each non-input gate v where f is the Boolean function associated to v and $v_1, \ldots, v_n$ are the gates with edges to v. The *constrained satisfiability problem* for Boolean circuits is formulated as follows: given that gates $c^+ \subseteq \mathcal{V}$ must be true and $c^- \subseteq \mathcal{V}$ must be false, is there a consistent valuation that respects these constraints, i.e., is there a satisfying valuation? The constrained Boolean circuit satisfiability problem is obviously **NP**-complete under the plausible assumption that each Boolean function in the system can be evaluated in polynomial time.

The encoding in the present work applies Boolean circuits where the following standard Boolean functions appear as gates: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\rightarrow$ (implication). In addition we use a function $\mathrm{cr}_L^U(v_1, \ldots, v_k)$ which is true in a valuation τ iff for the cardinality c of the set $\{\tau(v) = \text{true} \mid v \in \{v_1, \ldots, v_k\}\}$ holds that $L \leq c \leq U$ where L and U are fixed constants $0 \leq L \leq U$. The function cr_L^U represents actually a family of functions of which the following two forms are used in the paper: cr_0^1 (at most one true) and cr_1^1 (exactly one true).

4 Encoding

This section presents the structure of the Boolean circuits encoding the step and process executions of the synchronized product of n LTSs. For representational purposes the gates that appear are given certain illustrative names

briefly explained in Table 1. An in-depth description of them follows in subsequent sections with references to figures of gates drawn from the running example.

Table 1
Translation Predicates

Gate	Description
$ex(a,t)$	Action a is executed at time t, input gate.
$in(s,t)$	Execution is in state s at time t.
$sc(L,t)$	Component L scheduled at time t.
$ex(l,t)$	Transition l is executed at time t.
$uv(L,t)$	Unique visible transition from L at time t.
$ni(t)$	Disable idling at time t.
$enok(a,t)$	Execution of action a implies that it is enabled at time t.
$en(a,t)$	Action a is enabled at time t.

The encoding assumes that the LTSs do not have loops containing only τ-transitions involving more than one state. If that is the case, the corresponding component can be preprocessed so that the resulting LTS simulates all the executions of the original. The preprocessing step computes the maximal strongly connected components C_i of the LTS restricted to τ-transitions and replaces each C_i with a single state having as incoming and outgoing transitions the union of those in the set of states in C_i.

The representation follows certain conventions. The variable k is used to denote the length of the execution and the variables s, t, a and l are used to describe arbitrary states, positions in the execution, actions and transition labels, respectively. Based on the division of gates given in Sect. 3, the circuit is composed as follows. Firstly, some gates, namely those labelled with $ex(a,t)$ act as inputs. This special role is marked with two concentric circles. Secondly, the labels $ex(tr,t)$ and $sc(L,t)$ are attached to intermediate gates. Thirdly, the gates $uv(L,t)$ and $ni(t)$ are outputs constrained to true. This is reflected in the figures in which they appear by the symbol **T** appearing on the right side of the gate.

The gates labelled $in(s,t)$ can appear in different roles based on the value of t. Gates describing the initial states, i.e. $in(s,1)$ are inputs constrained to true and false depending on whether a state s is an initial state or not. For positions $1 < t \leq k$ the gates are intermediate and for the final position, i.e., $in(s,\ k\ +\ 1)$ they are output gates. When the translation scheme is augmented with a circuit detecting reachability properties, these gates are its inputs. The following subsections present the reasoning for all the gates and the section is concluded by a complete translation algorithm.

4.1 Control Flow

For encoding the control flow of the LTSs the idea is that the $in(s,t)$ gates serve to provide information regarding the progress of execution. For any initial state s of an LTS $in(s,1)$ is an input gate and is asserted true. This is in accordance with the fact that in the outset the execution in each component is in the initial states. In general, the execution may be in some state at time $t+1$ iff one of the following cases is true.

- The state was reached already at t and not left in step t.
- The state is reached due to it belonging to the τ-closure of some state reached via actions in step t.
- The state is reached by taking some of its incoming visible transitions in step t.

This provides the basis for the definition of the gate $in(s,t+1)$ an instance of which is needed for all the local states for all values $1 \leq t \leq k$. The structure of the gate is given in Fig. 2 for the state s_3 of the running example. It should be noted that τ-transitions from a state to itself can (and should) be ignored in the definition.

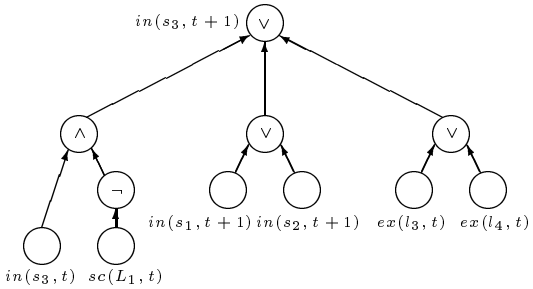

Fig. 2. Progress of Control Flow

The definition makes use of the $sc(L,t)$ and $ex(l,t)$ gates. The former captures the fact that a component L is scheduled iff a visible action in its alphabet is executed.

The reasoning behind the latter, the $ex(l,t)$ gate, is as follows. A transition is traversed in position t iff the action it is labeled with is executed in position t and the control flow is in its source state. It should be noted that the definition is not circular, but the control flow in position t together with the executed transitions define the control flow in position $t+1$. The picture on the right

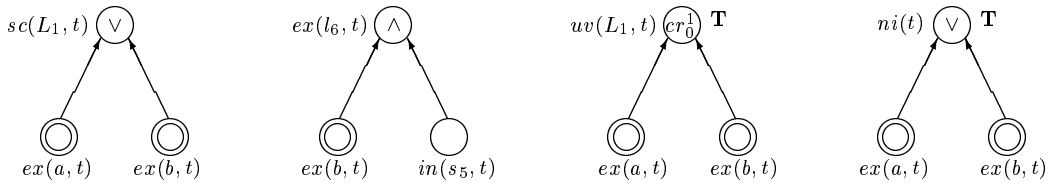

Fig. 3. Elements Illustrating Encoding from Running Example

in Fig. 3 illustrates the gate for the transition l_6 in the running example.

So far, the subcircuits presented have been definitions of the elements used in the encoding. To achieve correspondence with step and later process executions additional constraints need to be imposed. A step (process) execution has the property that only a single visible action is allowed to take place in a single component in each step. The arrangement to handle this is by using a cardinality constraint asserted true. An instance is given in Fig. 3 (the second one from the right).

The encoding may be further enhanced with a gate that disables idling. If such a gate is not added, the resulting circuit encodes step executions up to k whereas with it the executions are of precisely length k. Thus the gate limits the search space. As a downside short deadlocks may be missed if the verification process is started with too large a bound. Idling is disabled iff some visible action is executed, for the running example the gate is the rightmost in Fig. 3.

4.2 Synchronization

The synchronization of LTSs mandates that a visible action may be executed iff every LTS whose alphabet contains the action participates. So far, this has not been reflected in the subcircuits containing the input gates $ex(a,t)$. The condition is implemented by demanding that the executed action is enabled in each component having that label in its alphabet. An action a is enabled in a component iff it is in some state with an outgoing transition labelled a. The situation for the running example is illustrated in Fig. 4.

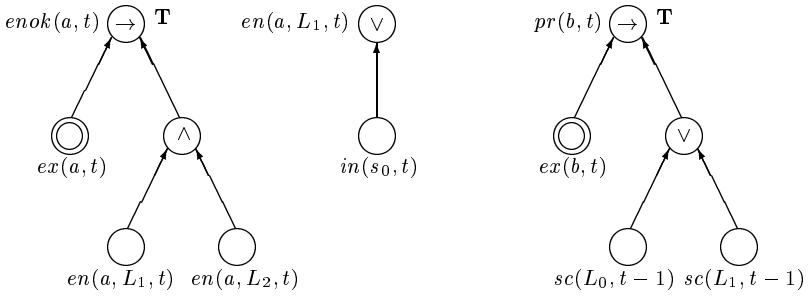

Fig. 4. Constraining the Input Gates (left, middle) and Enforcing Scheduling (right)

4.3 Translation Algorithm for Step Executions

Assume $L = (L_1 \| \cdots \| L_n)$ and a given bound k. Then the algorithm constructing a Boolean circuit encoding step executions of L of length k is as follows:

(i) To capture the requirement that each L_i is in the τ-closure of its initial states in V_1 add the gate $in(s,1)$ for all states s and constrain them to true if the above condition holds and false otherwise.

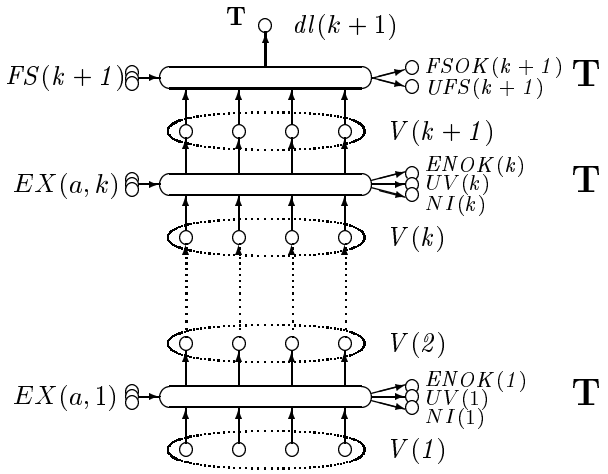

Fig. 5. Schematic Diagram of the Circuit

(ii) For all positions $1 \leq t \leq k$, add the following subcircuits:
  (a) For all states $s \in S_1 \cup \ldots \cup S_n$, include the subcircuit for $in(s, t+1)$.
  (b) For all the components L_j, add the subcircuit for $sc(L_j, t)$.
  (c) For all transitions with visible actions $l \in \Delta_1 \cup \ldots \cup \Delta_n$, add the subcircuit for $ex(l, t)$.
  (d) For each LTS L_j, add the subcircuit for $uv(L_j, t)$ and constrain it to true.
  (e) Add the circuit for $ni(t)$ and constrain it to true.
  (f) For all visible actions a, add the subcircuit for $enok(a, t)$ and constrain it to true.
  (g) For all components L_j, add the subcircuit for $en(a, L_j, t)$ for all its visible actions.

The structure of the circuit is schematically given in Fig. 5. In the bottom is the initial state $V(1)$ and on the top the last state $V(k+1)$ and a circuit for deadlock detection (introduced in Sect. 4.5). The unconstrained input gates appear on the left and the constrained outputs on the right, the labels capitalized to indicate that they denote several actual gates.

Let $SC(L, k)$ be the (step) circuit obtained by the translation algorithm. Given a satisfying truth valuation α for $SC(L, k)$ call an α-execution the execution $V_1 \stackrel{A_1}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{(k+1)}$ where the elements in each V_i are the states s with $\alpha(in(s, i)) =$ true and the elements in A_i the actions a having $\alpha(ex(a, i)) =$ true.

Theorem 4.1 *If the Boolean circuit $SC(L, k)$ has a satisfying truth valuation α, then there is an α-execution $V_1 \stackrel{A_1}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{k+1}$ which is a step execution.*

Theorem 4.2 *If $V_1 \stackrel{A_1}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{k+1}$ is a step execution of L, it is an α-execution for some satisfying valuation α of $SC(L, k)$.*

4.4 Process Executions

As can be seen from Definitions 2.4 and 2.9 the difference between step and process executions is rather simple. Indeed, the resulting circuit needs only one additional element. Namely, if an action is executed at $t+1$, then some participating component had to be scheduled in step t. On the right in Fig. 4 is an instance from the running example (executing action b in step 2).

The encoding algorithm needs the following addition for all $1 < t \leq k$.

(h) For all the visible actions $a \in \Sigma_1 \cup \ldots \cup \Sigma_n$ add the subcircuit $pr(a,t)$ and constrain it to true.

Figure 5 illustrates the circuit for step executions. Process executions would be modeled by adding the $PR(t)$ vector to the right hand side of the figure. Let $PR(L,k)$ be the (process) circuit obtained with the augmented algorithm.

Theorem 4.3 *If the Boolean circuit $PR(L,k)$ has a satisfying truth valuation α, then there is an α-execution $V_1 \stackrel{A_1}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{k+1}$ which is a process execution.*

Theorem 4.4 *If $V_1 \stackrel{A_1}{\rightarrow} \ldots \stackrel{A_k}{\rightarrow} V_{k+1}$ is a process execution of L, it is an α-execution for some satisfying valuation α of $PR(L,k)$.*

4.5 Reachability Properties

In Corollaries 2.8 and 2.11 it is stated that both step and process executions preserve the final states of the executions. Therefore, any state predicate concerning such a state can be studied with the presented approach.[4] A deadlock, i.e., a state with no outgoing transitions, is a particularly interesting case among such properties.

The synchronized product of LTSs can deadlock as a combination of two conditions. Firstly, components may end up in states with no outgoing transitions. Secondly, single components may indeed be able to proceed, but their synchronizing counterparts are in states where synchronization is not possible.

Thus a deadlock could be detected with circuits encoding such demands only based on the $in(s,k+1)$ gates. The former condition is simple, it can be detected by static analysis. The latter is more difficult to encode compactly. Therefore, deadlock detection is implemented by introducing a new input gate, $fs(s,L)$, for each component L and each state s with only visible outgoing actions.

The encoding is based on the reasoning that if there is a deadlocking interleaving execution, then the set of states reached in the associated step or

[4] There is a subtle issue which should be noted. The presented translation method assumes the following: if in a state s the state predicate to be studied holds then in all states reachable from s by using only τ-moves the predicate also holds, i.e. you cannot get out of a "bad" state by using only τ-moves. If some τ-moves do not respect this property, they must be converted to visible actions before the verification is started.

process execution reaches a state $V_{(k+1)}$ such that the deadlocking state s is be in $V_{(k+1)}$. The new input gate captures a single representative s' from each component L so that if the gate $fs(s', L)$ evaluates to true the state s' is the representative from the component L in $V_{(k+1)}$.

The gate has to be constrained in the following way. Firstly, an obvious soundness criterion is that a state has to be in $V_{(k+1)}$ for it to be a candidate. Secondly, to mandate the collection of final states to be a state of the interleaving executions, they have to be constrained to precisely one in each component. Instances from the component L_1 of the running example are given in Fig. 6. [5]

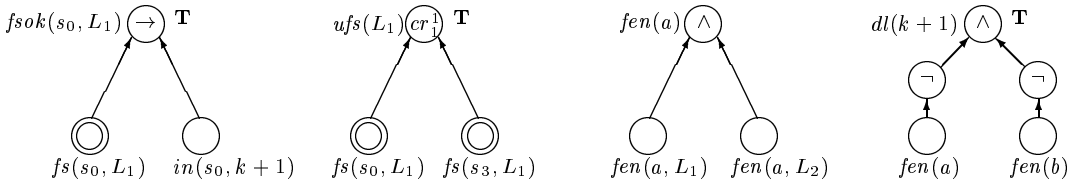

Fig. 6. State Predicate (Deadlock) Analysis

Having defined the fs gates a deadlock can be detected by the analysis of enabled actions in the final state. This is done by defining a $fen(a, L)$ gate for all the visible actions a and components L having the $fs(s, L)$ gates for state s in L with outgoing transitions labeled a as inputs. An action a is globally enabled in the final state iff it is enabled in all participating component iff the gate $fen(a)$ evaluates to true. Finally, a deadlock is a state where no action is globally enabled. The case for action a in the running example is illustrated on the right in Fig. 6 together with the deadlock detection gate for the entire example.

It should be noted that compared to the interleaving model, step and process executions may lose some of the intermediate states. However, it is not impossible to reason about them, provided that all state changes of interest to us can be observed through the occurrences of visible actions. The exact details of the following construction are left for further work, here just the main ideas are sketched. An additional component, called an observer automaton, can be added to the system. It observes the visible actions taking place by having all of them in its alphabet. Now any stuttering invariant safety property (which can be expressed as a regular language) can be reduced into the question of whether the observation automaton can reach a particular state. For the safety subset of LTL_{-X}, the linear temporal logic LTL without the next-time operator X, a finite automaton construction tool is available [14].

[5] This idea of guessing a final state combination can be used to compactly encode arbitrary state predicates.

5 Test Results

A set of test cases has been adopted from [7] taking those cases known to deadlock. The test cases are provided as LTS (fsa), Promela and SMV specifications thus rendering the comparison task feasible. The results of the tests are given in Table 2 with the following columns:

- Problem instance,
- St. k, bound for step executions, i.e., the smallest number of steps such that a deadlock is reached,
- St. s, running time for step executions as measured by `/usr/bin/time`,
- Pr. k and Pr. s, similarly for process executions,
- SMV k and SMV s, bound and time for NuSMV BMC [6],
- SMV bdd, running time for NuSMV BDD [5].

The tests were carried out with an AMD Athlon machine with a 1400 MHz CPU and 1 Gigabyte of memory running the Linux operating system. With the problem Dartes, no results could be obtained within a reasonable time limit (1 hour) using either NuSMV BMC or NuSMV BDD, therefore the entries are of the form N/A.

The results for the Boolean circuits were obtained by using a tool [13] to translate LTSs to Boolean circuits and then using the BCZCHAFF system [11] which first translates a circuit to CNF DIMACS form [11] and then solves it with zChaff version 2001.2.17 [16]. The fact that both the presented method and NuSMV BMC use zChaff as the back end adds credibility to their comparison. The running time for the step and process executions is the sum of generating the Boolean circuit from the specifications and solving it for the given bound. The running time for NuSMV BMC is composed of generating the CNF instance and solving if for exactly the given bound.

Even though the test cases do not have a lot of non-determinism it can be seen that the non-standard execution models compare favorably in terms of the bound and running time to those of NuSMV BMC. Compared to BDD-based model checking the results reiterate the fact that BMC is at its best in finding short deadlocks.

Experiments indicate that with these examples it sometimes takes zChaff far longer to prove a formula unsatisfiable than find a satisfying truth assignment with instances of comparable sizes. The phenomenon is most apparent in the example Key(4) where the time limit of one hour is exceeded with an unsatisfiable instance modeling process executions of length 29. The test cases and the tool translating LTSs to Boolean circuits are available for download at [13].

Table 2
Test Results (BCZCHAFF)

Problem	St. k	St. s	Pr. k	Pr.s	SMV k	SMV s	SMV BDD s
Dartes	31	2.0	31	0.53	N/A	N/A	N/A
DP(12)	1	0.028	1	0.028	> 8	830	0.12
Elev(1)	3	0.056	3	0.034	8	3.0	0.05
Elev(2)	5	0.16	5	0.11	11	3.1	0.17
Elev(3)	7	0.42	7	0.27	14	410	0.64
Elev(4)	9	1.6	9	0.74	17	120	2.7
Key(2)	35	510	35	200	> 30	2100	0.10
Key(3)	36	200	36	780	> 21	2700	0.27
Key(4)	37	10	37	11	> 19	3200	0.73
Key(5)	38	15	38	140	> 18	1900	3.2
Mmgt(3)	7	0.32	7	0.29	10	14	0.13
Mmgt(4)	8	0.77	8	0.35	12	73	0.25
Q(1)	9	0.25	9	0.25	> 11	1500	2.0
Hart(25)	50	1.2	50	0.71	51	7.0	0.12
Hart(50)	100	5.1	100	3.1	101	130	0.54
Hart(75)	150	12	150	7.6	151	990	1.9
Hart(100)	200	22	200	15	201	4800	5.5
Sentest(25)	33	0.63	33	0.7	38	4.2	0.12
Sentest(50)	58	2.1	58	2.3	63	40	0.45
Sentest(75)	83	4.5	83	5.1	88	220	1.5
Sentest(100)	108	8.0	108	8.3	113	980	4.6
Dac(15)	2	0.014	2	0.014	3	0.27	0.11
Speed(1)	4	0.038	4	0.030	7	0.13	0.07

6 Conclusions and Related Work

The paper studies bounded model checking of reachability properties of a system represented as a product of LTSs. Two nonstandard execution models, step and process executions, are proposed to capture sets of interleaving executions in a compact form.

The paper presents two translation schemes from an LTSs to Boolean circuits. In the first case, the resulting circuit encodes precisely the step executions of the product of LTSs under consideration and in the second the process executions. The encoding is compact leading to a circuit linear in the size of the bound k, more precisely $\mathcal{O}((\sum_j(|S_j|+|\Delta_j|+|\Gamma_j|))\cdot k)$ where S_j, Δ_j and Γ_j are the state space, transition relation and visible actions of LTS L_j, respectively. The encoding uses Boolean functions outside traditional propositional logic, namely cardinality constraints of the form cr_0^1 and cr_1^1, but the bound holds were the use of them disallowed. Such a function with indegree i can namely be simulated using $\mathcal{O}(i)$ new $\vee$, $\wedge$ and $\neg$ gates. The approach is backed by a set of test cases showing that the run times compare favorably to

a state-of-the-art interleaving BMC implementation in the NuSMV system.

The presented approach is considered only for models where the LTSs are presented explicitly. Translations from symbolical representations, like SMV models, is an interesting research problem for future work.

The idea for the paper arose as a comparison to the work done in [9]. The paper presents a BMC procedure to reachability check 1-safe Petri nets with step and process semantics. In addition to the different modeling formalism the approach is deterministic and does not exploit the inherent concurrency as effectively. The paper considers some of the same examples presented here. However, a direct comparison was omitted due to some inconsistencies in the state spaces of the fsa and 1-safe Petri net models. The differences could be traced to the fsa to 1-safe Petri net conversion performed in [15].

So far, only the verification of reachability properties has been considered, whereas LTL_{-X} model checking is left for future work. In [10] a translation of LTL_{-X} for step semantics is given using a logic programming approach.

Acknowledgement

The authors would like to warmly thank T. A. Junttila for creating the tool BCZCHAFF, and for fruitful discussions.

References

[1] 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*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[2] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Formal Methods in Computer Aided Design*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, November 2000.

[3] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer-Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer, 2001.

[4] B. Brard, Bidoit M., Finkel A, Laroussinie F., Petit A., Petrucci L., Ph. Schnoebelen, and McKenzie P. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer, 2001.

[5] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceeding of the 14th International Conference on Computer-Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, July 2002.

[6] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Integrating BDD-based and SAT-based symbolic model checking. In *Proceedings of 4th International Workshop on Frontiers of Combining Systems*, April 2002.

[7] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[8] V. Diekert and Y. Métivier. Partial commutation and traces. In *Handbook of formal languages, Vol. 3*, pages 457–534. Springer, Berlin, 1997.

[9] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory*, pages 218–232, August 2001.

[10] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming (TPLP)*, 2003. Accepted for publication. (CoRR: arXiv:cs.LO/0305040).

[11] T. A. Junttila. Boolean circuit tools, May 2003. `http://www.tcs.hut.fi/~tjunttil/circuits`.

[12] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for boolean circuit satisfiability testing. In *Computational Logic - CL 2000; First Internatinal Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567, London, UK, July 2000. Springer, Berlin.

[13] T. Jussila. A BMC tool translating LTSs to boolean circuits, May 2003. `http://www.tcs.hut.fi/~tjussila/otf`.

[14] Timo Latvala. Efficient model checking of safety properties. In *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[15] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceeding of 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 352–363. Springer, June 1997.

[16] M. Moskewicz, Y. Madigan, L. Zhao, Zhang L., and Malik S. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, July 2001.

Symbolic Computational Techniques for Solving Games

P. Madhusudan, Wonhong Nam and Rajeev Alur [1]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA.

Abstract

Games are useful in modular specification and analysis of systems where the distinction among the choices controlled by different components (for instance, the system and its environment) is made explicit. In this paper, we formulate and compare various symbolic computational techniques for deciding existence of winning strategies. The game structure is given implicitly, and the winning condition is of the form "*p until q*" for state predicates p and q. The first technique employs symbolic fixpoint computation using ordered binary decision diagrams [8]. The second technique checks for the existence of strategies that ensure winning within k steps, for a user specified bound k, by reduction to the satisfiability of quantified boolean formulas. Finally, the bounded case can also be solved by reduction to satisfiability of ordinary boolean formulas, and we discuss two techniques, one based on encoding the strategy tree, and one based on encoding a witness subgraph, for reduction to SAT. We compare the various approaches on two examples using existing tools such as MUCKE [7], MOCHA [3], SEMPROP [16], QUBE [13], BERKMIN [12].

1 Introduction

The motivation for solving games in formal analysis originated with Church's synthesis problem in the context of automatically synthesizing circuits from specifications [10]. Games have since then become popular in formal methods with various applications including control of discrete event systems [22], realizability and synthesis, and model-checking μ-calculus formulae [24]. In formal verification, they have several applications in verifying reactive systems where the agents comprising the system are viewed as players of a game: in modular verification [15], in synthesis of formal interfaces to modules [9] and in approaches to compositional verification [1,2].

[1] Email: {madhusudan, wnam, alur}@cis.upenn.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

Research and related applications have led to variety of game formulations such as infinite games on finite graphs, concurrent multi-player games and games on pushdown systems [24]. However, the simplest game that most solutions computationally rely on is the two-player reachability game on a finite graph. Such a game is played between two players, the *system* and the *environment*, and the game problem is to check whether the system has a winning strategy that will force the game from the initial position to some goal position, no matter how the environment plays.

Though the theoretical complexity of solving various games in the literature is well understood, there has been relatively less effort spent in identifying how the powerful symbolic techniques used in model-checking fare in solving games with large state-spaces. In this paper, we initiate such an effort by a comparative and experimental study of solving simple reachability games (augmented with a safety condition) using techniques that use Bdds, Sat-solvers and Qbf-solvers. We model games symbolically using boolean variables and succinct boolean expressions describing the transitions — the explicit game this defines would be typically exponential in the size of the definition.

The standard attractor-set approach to solve reachability games is a simple fix-point algorithm that can easily be implemented using Bdds. There are two kinds of Bdd-based solvers we use: Mocha which is a model-checker that can directly handle specifications in a game logic called alternating temporal logic (Atl) and Mucke which is a μ-calculus model checker especially tuned and extended to handle μ-calculus formulas.

For propositional solvers, we consider bounded reachability games. We first consider games where we ask whether the system has a strategy that will ensure the game reaches the goal within k steps, where k is a user-specified parameter. The natural way to encode this as a propositional satisfiability problem is using a quantified boolean formula, where there is a prefix of alternating quantifiers of length $2k$ that capture a strategy for the system followed by a boolean formula that checks whether the strategy is indeed winning for the system. We then use Qbf solvers Semprop [16], Quaffle [25] and Qube [13] to solve these formulas.

In recent years, there has been a significant interest in engineering Sat-solvers that has resulted in very efficient solvers, while the effort in speeding up Qbf solvers has been relatively less. We hence also consider encodings of games into Sat problems, in two different ways. In the first approach, we use Sat to guess a winning strategy tree of depth k (the tree is exponential in k). This can be seen essentially as "unwinding" the alternating quantification in the Qbf formula above into a tree of existential quantifications, by converting each universal choice to all possible choices. We hence get an exponential-sized Sat formula which is satisfiable if and only if there is a strategy that wins in k steps, and we use the Sat-solvers zChaff [19] and Berkmin [12].

In the strategy tree guessed above, several nodes of the tree could represent the same position in the game and the tree encodes the strategies from these

nodes separately. Since reachability games have zero-memory strategies, we need not guess separate strategies from these nodes. In the second reduction to Sat, we consider a variation where we essentially guess a *directed acyclic graph* of positions of the game which encode a strategy for the system and which witnesses the fact that the system wins the game. Given a parameter n on the size of such a witness set, our reduction checks whether the system has a strategy such that there is a set of positions bounded by n within which the system can force the game to be within and reach the goal. This is perhaps the more natural generalization of bounded model-checking to games.

We compare all the above methods and the different encodings described above using two examples that can be scaled. The first example is a pursuer-evader game where the objective is to guide a robot from one end of a grid to another while evading another slower robot that moves arbitrarily in the grid. Since our results show that Bdd methods outperform both Sat and Qbf methods by a large margin for this example, we consider in the second example a game which is known to be hard for Bdds (using the *swap* example from [18]). However, it turns out that Bdds still outperform the Sat and Qbf methods. We postpone a more detailed discussion of the results to the concluding section.

Our aim in this paper is to have a common platform to specify symbolic games so as to compare various symbolic techniques and evaluate them. The games we consider involve continuous interaction between the two players, as is common in most games studied in formal methods. The use of symbolic methods to solve problems related to games is not new. Symbolic methods have been proposed and studied in the area of planning in AI, for example, in conditional planning using Qbf methods [21] and for universal planning using Bdds [14] (see also [5]). However, we do not know of any comparative study of solving games using different symbolic approaches.

The paper is organized as follows. Section 2 lays out the precise definition of symbolic two-player reachability games. In Section 3 we outline two approaches using Bdds to solve games, one using Atl specifications in Mocha and the other using μ-calculus specifications in Mucke. Section 4 deals with solving bounded versions of the game problem, using reductions to satisfiability of Qbf and Sat formulas. For the Sat reduction we outline both the strategy-tree approach as well as the witness-graph approach. We present our experimental results for two game examples in Section 5 and conclude in Section 6.

2 Games

In this section, we define the required terminology. Let X be a finite set of variables. We write $X' = \{x' |\ x \in X\}$ for the set of primed variables of X. We denote by $Val(X)$ the set of all total functions that map every variable in X to a value in its domain. The set of all predicates over X is denoted by $\mathcal{P}(X)$. Given $p \in Val(X)$ and a predicate φ over $X = \{x_1, \cdots, x_n\}$, we write $\varphi[p] = \varphi[x_1/p(x_1), \cdots, x_n/p(x_n)]$ for the truth value obtained by replacing

each variable $x_i \in X$ in φ with the value $p(x_i)$.

We model a *game* [24] between a *system* and its *environment* using a *game structure* $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ with the following components:

- X_S is a finite set of *variables* the system controls, and X_E is a finite set of *variables* the environment controls with $X_S \cap X_E = \emptyset$. We write $X = X_S \cup X_E$ for the set of system and environment variables, and $Q = Val(X)$ for the set of states of S.
- M_S is a finite set of *move* variables [2] which determine the next *move* of the system and M_E is a finite set of *move* variables which determine the next *move* of the environment. We assume $M_S \cap M_E = \emptyset$, $M_S \cap X = \emptyset$ and $M_E \cap X = \emptyset$.
- $T_S \in \mathcal{P}(X, M_S, X'_S)$ is a *transition predicate* for the system variables. For each $q \in Val(X)$, $m_S \in Val(M_S)$ and $q'_S \in Val(X'_S)$, if $T_S[q, m_S, q'_S] = true$ then q'_S is the next valuation of variables in X_S when the system picks the move m_S at the state q. Similarly, $T_E \in \mathcal{P}(X, M_E, X'_E)$ is a *transition predicate* for the environment variables. For each $q \in Val(X)$, $m_E \in Val(M_E)$ and $q'_E \in Val(X'_E)$, if $T_E[q, m_E, q'_E] = true$ then q'_E is the next valuation of variables in X_E when the environment picks the move m_E at the state q.

Now, we define a *game* $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with a game structure S, an *initial state* [3] $I \in Val(X)$, a *goal predicate* $\mathcal{G} \in \mathcal{P}(X)$ and a *safe predicate* $\mathcal{S} \in \mathcal{P}(X)$ where for each $q \in Val(X)$, if $\mathcal{G}[q] = true$ then the state q is in the *goal* region, and if $\mathcal{S}[q] = true$ then the state q is in the *safe* region. The game starts in the initial state and in every step, the system and the environment pick a move simultaneously and the state evolves according to this choice. If the goal region is reached then the system wins. If the current state is not in the safe region, the environment wins. Otherwise, the game continues forever.

For two states p and q, we say that q is the *successor* of p if there are $m_S \in Val(M_S)$ and $m_E \in Val(M_E)$ such that $T_S[p, m_S, q'_S] = true$, $T_E[p, m_E, q'_E] = true$ and $q = q_S \cup q_E$. We assume that there exists at least one successor at every state. A *path* of S is a finite or infinite sequence $\lambda = q_0, q_1, \cdots$ of states such that for all positions $i \geq 0$, q_{i+1} is a successor of q_i. For a path λ and a position $i \geq 0$, we use $\lambda[i]$ and $\lambda[0, i]$ to denote the i-th state of λ and the finite prefix $q_0, q_1, \cdots, q_i$ of λ, respectively. A *strategy* for the system is a function $f : Q^+ \rightarrow Val(M_S)$ which maps every nonempty finite state sequence $\lambda \in Q^+$ to a move $f(\lambda) \in Val(M_S)$. Given a strategy f, we define the *plays* of f to be the set $plays(f)$ of paths which are possible when the system follows the strategy f; that is, a path $\lambda = q_0, q_1, \cdots$ is in $plays(f)$ if for all positions $i \geq 0$, there are $m_S \in Val(M_S)$ and $m_E \in Val(M_E)$ such that $m_S = f(\lambda[0, i])$ and q_{i+1} is the $\langle m_S, m_E \rangle$ successor of q_i. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, a strategy

[2] In many examples, M_S and M_E will contain a single variable but in general, if a system has multiple components then there can be a move variable for each component.

[3] We can handle multiple start states by introducing a new state as an initial state with moves to all the start states.

f is a *winning strategy* in the game G if for all $\lambda = q_0, q_1, \cdots \in \mathit{plays}(f)$ such that $q_0 = I$, there exists a position $i \geq 0$ such that $\mathcal{G}[q_i] = \mathit{true}$ and for all positions $0 \leq j < i$, $\mathcal{S}[q_j] = \mathit{true}$. Finally, given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, the *game problem* is to check whether the system has a winning strategy in the game G.

Example 1.
Consider the game between an *evader* E and a *pursuer* P on an $n \times n$ grid as shown in Figure 1. The evader tries to reach the predefined *goal* position G without being caught by the pursuer. The evader chooses one amongst five moves: *up, down, left, right* and *stay* in every step. The pursuer, however, chooses one such move only in every odd step and it must stay stationary in every even step. Considering the evader as the system player and the pursuer as the environment player, we can define the game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ as follows.

First, we model the game structure by $S = (X_S, X_E, M_S, M_E, T_S, T_E)$.

- $X_S = \{x_e, y_e\}$ where x_e and y_e ranging over $\{0, \cdots, n-1\}$ are the x-y coordinates of the evader, and $X_E = \{x_p, y_p, \mathit{clock}\}$ where x_p and y_p ranging over $\{0, \cdots, n-1\}$ are x-y coordinates of the pursuer and *clock* ranging over $\{0, 1\}$ is a toggle specifying when the pursuer can change its position.
- $M_S = \{m_e\}$ and $M_E = \{m_p\}$ where m_e and m_p range over $\{\mathit{up}, \mathit{down}, \mathit{left}, \mathit{right}, \mathit{stay}\}$.
- $$T_S \equiv \begin{bmatrix} ((x_e > 0) \wedge (m_e = \mathit{left}) \wedge (x'_e = x_e - 1) \wedge (y'_e = y_e)) \\ \vee((x_e < n-1) \wedge (m_e = \mathit{right}) \wedge (x'_e = x_e + 1) \wedge (y'_e = y_e)) \\ \vee((y_e < n-1) \wedge (m_e = \mathit{up}) \wedge (x'_e = x_e) \wedge (y'_e = y_e + 1)) \\ \vee((y_e > 0) \wedge (m_e = \mathit{down}) \wedge (x'_e = x_e) \wedge (y'_e = y_e - 1)) \\ \vee((m_e = \mathit{stay}) \wedge (x'_e = x_e) \wedge (y'_e = y_e)) \end{bmatrix}$$

$$T_E \equiv \begin{bmatrix} \Big((\mathit{clock} = 1) \wedge \big(((x_p > 0) \wedge (m_p = \mathit{left}) \wedge (x'_p = x_p - 1) \wedge (y'_p = y_p)) \\ \qquad \vee((x_p < n-1) \wedge (m_p = \mathit{right}) \wedge (x'_p = x_p + 1) \wedge (y'_p = y_p)) \\ \qquad \vee((y_p < n-1) \wedge (m_p = \mathit{up}) \wedge (x'_p = x_p) \wedge (y'_p = y_p + 1)) \\ \qquad \vee((y_p > 0) \wedge (m_p = \mathit{down}) \wedge (x'_p = x_p) \wedge (y'_p = y_p - 1))\big)\Big) \\ \vee((m_p = \mathit{stay}) \wedge (x'_p = x_p) \wedge (y'_p = y_p)) \end{bmatrix}$$

$$\wedge(\mathit{clock}' \neq \mathit{clock}).$$

$I \equiv \{x_e = 0, y_e = 0, x_p = 1, y_p = 3\}$ if the initial position of the evader is $(x = 0, y = 0)$ and the initial position of the pursuer is $(x = 1, y = 3)$. $\mathcal{G}$ is *true* if the x-y coordinates of the evader are same with the predefined goal position. $\mathcal{S}$ is *true* if the x-y coordinates of the evader are different from the pursuer's: $\mathcal{S} \equiv (x_e \neq x_p) \vee (y_e \neq y_p)$.

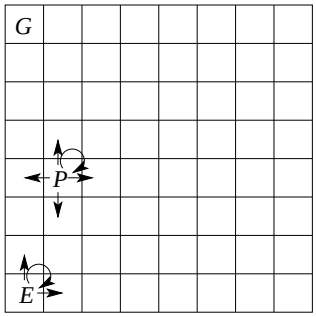

Fig. 1. Pursuit-evasion Game

Algorithm [Symbolic model checking for game problems]

Input: a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$.
Output: the answer to the model checking problem for the game.

$\rho := \mathit{False}$;
$\tau := \mathcal{G}$;
while $\tau \not\equiv \rho$ **do**
    $\rho := \rho \vee \tau$;
    $\tau := \mathit{Pre}^G(\rho) \wedge \mathcal{S}$;
od;
if $\rho(I)$ **then** return *true*;
else return *false*;

Fig. 2. Symbolic model checking algorithm for game problems

3 Solving games using BDDs

In this section, we solve games using binary decision diagrams (Bdds). The standard attractor-set method to solve games is a fix-point algorithms that can be implemented using Bdds. Figure 2 shows a symbolic model checking algorithm for our game problem, which manipulates state sets of S. Given a goal region and a safe region, we compute all states from which there is a winning strategy for the system. Note that the function Pre^G is different from the pre-image function of Ctl model checkers. The function Pre^G, when given a predicate $\rho(X_S, X_E)$, returns a predicate $\mathit{Pre}^G(\rho) \in \mathcal{P}(X)$ for the set of states p such that from p, the system enforces the next state to satisfy ρ no matter how the environment behaves. Formally,

$$\mathit{Pre}^G(\rho) \equiv \exists M_S, X'_S. \forall M_E, X'_E.\ T_S(X, M_S, X'_S) \wedge (T_E(X, M_E, X'_E) \rightarrow \rho(X'_S, X'_E)).$$

In the algorithm, sets of states and the transition relation are represented by Bdds [8]. Both the Atl model checker and μ-calculus model checker use this algorithm.

ATL Model Checking

Mocha[3] is a verification environment for modular verification for alternating-time temporal logic, which is a game logic extension of Ctl.

Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, we specify a game structure S as reactive modules [3] where the system and its environment are described in separate modules, and specify $\mathcal{G}$ and $\mathcal{S}$ as an Atl formula using the *until* operator $\mathcal{U}$. The logic Atl admits a formula $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$, where ϕ and ψ are state predicates and A is a subset of players. The formula $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$ asserts that the players in A can cooperate to keep satisfying ϕ until satisfying ψ no matter how the remaining players behave. Considering A as the system, the semantics of $\langle\langle A \rangle\rangle \phi \, \mathcal{U} \psi$ is exactly same as the game problem. For Example 1, we specify the evader and the pursuer as separate modules, and specify the game property as the Atl specification, $\langle\langle Evader \rangle\rangle (safe \, \mathcal{U} \; goal)$. Then, we use symbolic Atl model checking of Mocha, which implements the algorithm shown in Figure 2.

μ-calculus Model Checking

The μ-calculus [4] is propositional modal logic extended with the least fixpoint operator and is interpreted over Kripke structures. While μ-caculus model-checking can be seen to be equivalent to evaluating *infinite parity games* on finite graphs, the μ-calculus also trivially encodes solutions to reachability games. In our context, the μ-calculus formula:

$$\mu X.(goal \vee (safe \wedge \bigvee_{m_s \in Val(M_S)} \bigwedge_{m_e \in Val(M_E)} \langle m_s, m_e \rangle X))$$

computes the winning area for player S, as it stands for the least set X containing the goal configurations as well as those configurations from which the system can force a move into X.

Since least fixpoint computations can be performed symbolically, we can use symbolic μ-calculus model checkers to solve games using Bdds. The model-checker we consider is Biere's model checker Mucke (μcke) [7], which is developed with an aim to be a μ-calculus model checker that performs as well as symbolic model-checkers like Smv on the Ctl fragment. Mucke is a Bdd-model checker optimized for the μ-calculus using techniques similar to those employed in model-checkers for Ctl (like allocating fixed variable orderings for variables computing fixpoints, frontier set simplification, etc.).

When coding games into μ-calculus, we can also implement *early termination*, i.e. terminating the above fix-point computation when we reach an initial state. This can be encoded as:

$$\mu X.(goal \vee (\exists \bar{x} \in X : I\bar{x}) \vee (safe \wedge \bigvee_{m_s \in Val(M_S)} \bigwedge_{m_e \in Val(M_E)} \langle m_s, m_e \rangle X))$$

In the above, if an initial state is reached, the set X immediately gets set to the entire set of states and the fixpoint terminates.

4 Solving Bounded Games

Symbolic model checking [17] has been acknowledged as an efficient verification technique. Many symbolic model checkers use Bdds [8] as representations for sets of states and transition relation. However, the size of Bdds may increase exponentially as the number of variables.

Recently, a new type of model checking technique, *bounded model checking* with satisfiability solving [6,11], has led to promising results. In bounded model checking, given a transition system S, a temporal logic formula f and a user-supplied bound $k \in \mathbb{N}$, we construct a propositional formula $[\![S, f]\!]_k$ which is satisfiable if and only if the formula f is valid along some path of length k. Then, we solve the formula $[\![S, f]\!]_k$ using a Sat solver.

4.1 QBF Methods

For solving bounded games, we need more definitions. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$, a strategy f and a bound k, $plays_k(f)$ is the set of plays of length k which are possible when the system follows the strategy f. A strategy f is a *k-winning strategy* in a game G if for all $\lambda = q_0, \cdots, q_k \in plays_k(f)$ are winning, i.e., there exists a position $0 \leq i \leq k$ such that $\mathcal{G}[q_i] = true$ and for all positions $0 \leq j < i$, $\mathcal{S}[q_j] = true$. The *bounded game problem* is, given a game G and a bound k, to check whether the system has a k-winning strategy in the game G. Consequently, we want to construct a boolean formula $\Phi^1_{G,k}$ which is satisfiable if and only if the system has a k-winning strategy in the game G.

Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ and a bound k, we denote, for every $0 \leq i \leq k$, the i-th copy of X, X_S, X_E by X^i, X^i_S, X^i_E, respectively. we divide I into I_S and I_E which are the initial values for X_S and X_E, respectively. However, unlike bounded model checking, we need alternations of existential quantification and universal quantification in order to solve a bounded game problem. Therefore, the formula $\Phi^1_{G,k}$ is a quantified boolean formula beginning with a prefix $\exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^{k-1}_S, M^{k-1}_S . \forall X^{k-1}_E, M^{k-1}_E . \exists X^k_S . \forall X^k_E$. $\Phi^1_{G,k}$ describes that there exists a series of system's moves to guarantee that for all series of environment's moves, the goal region is reached through the safe region as long as the environment proceeds according to the transition relation. $\Phi_{G,k}$ is as follows.

$$\Phi^1_{G,k} \equiv \exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^{k-1}_S, M^{k-1}_S . \forall X^{k-1}_E, M^{k-1}_E . \exists X^k_S . \forall X^k_E .$$
$$I_S(X^0_S) \wedge \phi_1 \wedge \Big((I_E(X^0_E) \wedge \psi_1) \rightarrow \rho \Big)$$

where,

- $\phi_1 \equiv \bigwedge_{i=0}^{k-1} T_S(X_i, M^i_S, X^{i+1}_S)$,
- $\psi_1 \equiv \bigwedge_{i=0}^{k-1} T_E(X_i, M^i_E, X^{i+1}_E)$ and

- $\rho \equiv \bigvee_{i=0}^{k} (\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j))$.

In the above formula $\Phi^1_{G,k}$, the subformulas, ϕ_1 and ψ_1, force that the next states along the path should obey the transition relation and ρ encodes reachability to the goal region through the safe region within k steps.

The total number of variables in $\Phi^1_{G,k}$ is $O(k \cdot N)$ where $N = |X \cup M_S \cup M_E|$, and the length of $\Phi^1_{G,k}$ (after some simplification) is $O(k \cdot (|T_S| + |T_E| + |\mathcal{G}| + |\mathcal{S}|) + |I_S| + |I_E|)$ where $|\mathcal{G}|, |\mathcal{S}|, |T_S|, |T_E|, |I_S|$ and $|I_E|$ are the lengths of formulas. In this expression, $k \cdot (|T_S| + |T_E|)$ is the dominant factor because $|T_S|$ and $|T_E|$ are quadratic in N, but $|\mathcal{G}|$ and $|\mathcal{S}|$ and is linear in N.

We define a new formula $\Phi^2_{G,k}$ which has three extra copies of the variables $X \cup M_S \cup M_E$, but which is shorter than the previous formula $\Phi^1_{G,k}$ since it has only one occurrence of T_S and T_E. The trick is to have an additional universal quantification after the k alternating quantifiers and to treat these as temporary variables and check that if they match the i^{th} and $(i+1)th$ copy of the original variables, then they satisfy the predicates T_S and T_E. Subsequently, the total number of variables in $\Phi^2_{G,k}$ is $O(k \cdot N)$ and the length of $\Phi^2_{G,k}$ (after some simplification) is $O(k \cdot (|\mathcal{G}| + |\mathcal{S}|) + |T_S| + |T_E| + |I_S| + |I_E|)$. $\Phi^2_{G,k}$ is given by:

$$\Phi^2_{G,k} \equiv \exists X^0_S, M^0_S . \forall X^0_E, M^0_E . \cdots . \exists X^k_S . \forall X^k_E . \forall Y, Y_M, Y', Z, Z_M, Z' .$$
$$I_S(X^0_S) \wedge \phi_2 \wedge \Big((I_E(X^0_E) \wedge \psi_2) \rightarrow \rho \Big)$$

where,

- $\phi_2 \equiv \bigvee_{i=0}^{k-1} ((X^i = Y) \wedge (M^i_S = Y_M) \wedge (X^{i+1}_S = Y')) \rightarrow T_S(Y, Y_M, Y')$,
- $\psi_2 \equiv \bigvee_{i=0}^{k-1} ((X^i = Z) \wedge (M^i_E = Z_M) \wedge (X^{i+1}_E = Z')) \rightarrow T_E(Z, Z_M, Z')$ and
- $\rho \equiv \bigvee_{i=0}^{k} (\mathcal{G}(X^i) \wedge \bigwedge_{j<i} \mathcal{S}(X^j))$.

We denote by *M1* the method which uses the first formula $\Phi^1_{G,k}$ to solve the game, and by *M2* the method which uses $\Phi^2_{G,k}$. We use Qbf solvers such as Semprop [16], Quaffle [25] and Qube [13] in order to solve the above quantified boolean formulas.

4.2 SAT Method using Strategy Tree

The bounded game problem is naturally translated to a Qbf solving problem as we saw in Section 4.1 and we must use Qbf solvers. However, several Sat solvers have recently shown promising results. In the next two subsections, we show how to translate the game problem to a boolean formula only with existential quantification in order to use Sat solvers.

For translating the quantified formula for $\Phi_{G,k}$ in the previous section into a boolean formula, we need to eliminate universal quantification by introducing extra copies of variables in order to specify explicitly all cases without universal quantification; for example, $\forall x . \exists y . (x \wedge y) \equiv \exists y_1, y_2 . ((\mathit{true} \wedge y_1)) \wedge (\mathit{false} \wedge y_2))$. Figure 3 shows relations between successors and predecessors in Qbf and

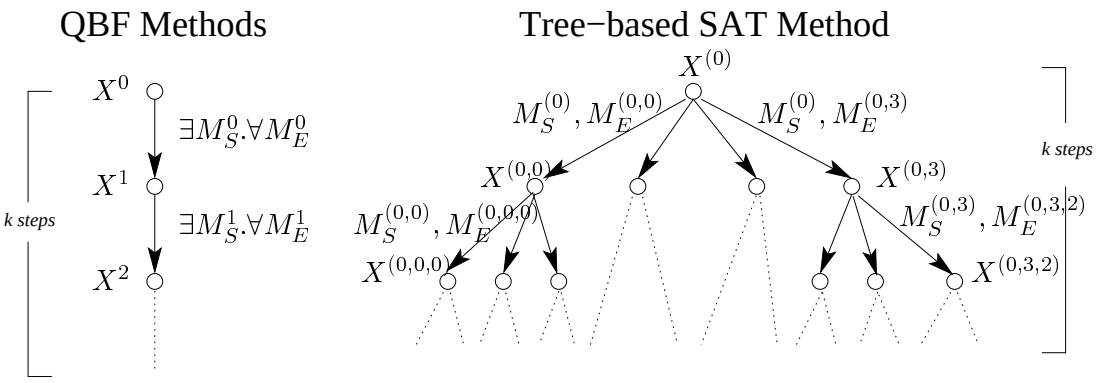

Fig. 3. Tree-based Sat Method

Sat methods. In tree-based Sat method, we introduce explicitly one copy of variables for every node in the tree. Thus, the number of copies is exponential in the bound k.

Every path of the tree-based Sat method corresponds to a play of length k and we just need to write a formula to check that the paths stay in the *safe* region until they reach the *goal* region, which we write as the formula $\Phi^3_{G,k}$.

The number of variables in $\Phi^3_{G,k}$ is $O(N \cdot m^{k-1})$ where m is the maximum number of environment's moves and the length of $\Phi^3_{G,k}$ is $O(m^{k-1} \cdot (|T_S| + |T_E| + k \cdot |\mathcal{S}| + k \cdot |\mathcal{G}|) + |I_S| + |I_E|)$. We then use Berkmin [12] and zChaff [19] in order to solve the boolean formula $\Phi^3_{G,k}$.

4.3 SAT Method using Witness Set

In the strategy-tree based Sat method, we constructed a tree which is a witness for a bounded game problem with a bound k. The tree, however, could have many identical states and we check the strategy from the identical states many times. In this section, we introduce a method that can generate a witness set with less copies of variables. The main idea is to construct a set that witnesses the fact that the system wins. Thus, given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ and a user supplied $n \in \mathbb{N}$, we generate a boolean formula $\Phi^4_{G,n}$ which is satisfiable if and only if we can generate a witness set with n states. First, we define $T_i(X, M_S, X')$ as a predicate for the next state when the environment's move is fixed. For each element m_i of the set $\{m_1, m_2, \cdots, m_{max}\}$ of the environment's moves, $T_i(X, M_S, X')$ is the predicate obtained from $T_S(X, M_S, X'_S) \wedge T_E(X, M_E, X'_E)$ (where $X' = X'_S \cup X'_E$) by replacing each of the variables $v \in M_E$ with the value $m_i(v)$. Now, we define a witness set for a bounded game problem as follows. Given a game $G = \langle S, I, \mathcal{G}, \mathcal{S} \rangle$ with $S = (X_S, X_E, M_S, M_E, T_S, T_E)$ and user supplied number n, $W = \{q_1, q_2, \cdots, q_n\}$ is a witness set for the game G if and only if

- for the initial predicate I of G, $I[q_1] = \mathit{true}$, and

- for each $q_i \in W$, $\mathcal{G}[q_i] = \mathit{true}$, or, $\mathcal{S}[q_i] = \mathit{true}$ and there exists a system move m_S^i such that for each valid element m_j in the set $\{m_1, m_2, \cdots, m_{max}\}$ of environment moves at q_i, there exists $i < l \leq n$ such that $T_j[q_i, m_S^i, q_l] = \mathit{true}$.

The formula $\Phi^4_{G,n}$ for witness-based Sat method is as follows.

$$\Phi^4_{G,n} \equiv I(X^1) \wedge \bigwedge_{i=1}^{n} \Big(\mathcal{G}(X^i) \vee \Big(\mathcal{S}(X^i) \wedge \bigwedge_{j=1}^{max} (V_j(X^i) \rightarrow \bigvee_{l>i}^{n} T_j(X^i, M_S^i, X^l)) \Big) \Big)$$

where $V_j \in \mathcal{P}(X)$, for every $1 \leq j \leq max$, is a *validity predicate* for the environment: $V_j[q] = \mathit{true}$ if and only if m_j is valid environment's move at the state q. The definition of a witness set forces that every copy q_i which is not a goal, must have a transition to some q_l where l is strictly larger than i. Note that q_n must hence be a goal position and in fact the definition forces all plays encoded in the witness set to end in the goal. In the formula $\Phi^4_{G,n}$, the total number of variables is $O(n \cdot N)$ and the length of the formula is $O(mn^2 \cdot |T_j| + n \cdot (|\mathcal{G}| + |\mathcal{S}|) + |I|)$ where m is the maximum number of environment's moves. We again use Berkmin and zChaff in order to solve the boolean formula $\Phi^4_{G,n}$.

5 Experimental Results

We also consider a second example, which is known to be hard for Bdds [18].

Example 2.
The second example is *swap* introduced in [18]. We change the example into a game problem. There is an array $A[\ ]$ with n elements which are m-bit binary numbers. We assume that $n \leq 2^m$ so that all elements in the array can be distinct. Initially we have, for all $0 \leq i < n$, $A[i] := i$. At each step, the system chooses a direction between *left* and *right* and the environment chooses an index i, in the range $0, \cdots, n-1$; then the value of $A[i]$ is swapped with that of $A[(i-1) \bmod n]$ or $A[(i+1) \bmod n]$, according to the direction the system picked. The property we want to check is whether the system can eventually make $A[0]$ and $A[1]$ same no matter what the environment does (the system clearly loses).

We compare the methods we addressed using the Examples 1 and 2. For Qbf methods, our program first generates a Boolean circuit [23] file, which is a more succinct format than Cnf. Then we use bc2cnf [23] to translate the Boolean circuit into Cnf. In the process, many intermediate variables are introduced. Finally, our program attaches quantification to the Cnf file automatically and we use Qbf solvers such as Semprop, Quaffle and Qube to solve the Cnf formula with quantification.

Also, for Sat methods we generate a Boolean circuit file and translate it to Cnf using bc2cnf. We use the Sat solvers Berkmin and zChaff on the Cnf formula. All experiments were performed on a PC using a 1GHz Pentium III processor, 1.5GB memory and the Linux operating system.

The results for Example 1 are shown in Table 1 where the time shown is the execution time in seconds, '–' means that it did not complete in 10 hours, and * means the size of the input file was too large to execute (over 1GB). In Bdd methods, the number in parenthesis is the number of iterations taken to reach the fix-point while in the witness method, the number in parenthesis is the size of the witness set. For early termination results, the goal position was chosen as $(n/3, 3n/4)$ for the $n \times n$ grids. In this example, Mucke performed better than Mocha. For Qbf method $M1$, Qube (Ver. BJ1.0) worked best and for Qbf method $M2$, Semprop (Ver. 240202) showed the best result. For Sat methods, Berkmin worked best. The results in the table are the results for the tools that performed best. For this example, Bdd-based methods seem better than Qbf, and Sat-based methods seem better than Qbf-methods.

Table 2 shows the results for Example 2 where the Bdd method outperformed Qbf and Sat methods. Unlike Example 1, the Qbf method was better than the tree-based Sat method. This is perhaps because, in Example 2, the environment has n moves at every stage, which makes the strategy tree very large, while in Example 1, it has at most five moves at any stage.

6 Conclusions

We have presented various symbolic methods using Bdds, Sat-solvers and Qbf-solvers to solve symbolically presented succinct games and evaluated them on two examples. This research is preliminary and one cannot draw hard conclusions yet. From the current results, however, it does seem that Bdds (especially Mucke) outperform methods that use propositional solvers. The main problem with reduction to Sat seems to be the exponential blow-up in the reduction to game witnesses. Also, just reducing the size of the formula by making it more complex, seems to make Sat and Qbf solvers perform worse than with a simple but larger encoding. If one could come up with a very small notion of a witness set for winning games, the propositional solvers may turn out to be more powerful.

There are several issues that are interesting for future study. First, most applications require to solve *partial information games* and it is not clear how to extend the methods to handle this. Also, once we know that the system indeed wins the game, we do not know how hard it is to extract a winning strategy of reasonable size from the above procedures.

Games have been recently used in the extraction of formal interfaces to software modules, in order to check consistency between software components [9]. It would be interesting to try out the above symbolic game solving techniques in such a domain.

Grid size	Bdd methods			Bounded methods				
	Mocha	Mucke		Step(k)	Qbf methods		Sat methods	
		Normal	Early		$M1$	$M2$	Tree	Witness
4×4	0 (13)	3 (7)	3 (3)	4 6 7 15 16	1 193 2354 – –	550 – – – –	0 0 0 17 *	818 (25)
8×8	6 (21)	3 (16)	3 (16)	4 5 15 16	125 34783 – –	– – – –	0 0 117 *	–
16×16	190 (36)	3 (32)	3 (32)	12 15 16	– – –	– – –	29 135 *	–
32×32	6493 (68)	5 (64)	5 (64)	12 15 16	– – –	– – –	58 531 *	–
256×256	–	373 (512)	100 (263)	8 12	– –	– –	– –	–
512×512	–	–	4024 (517)	8 12	– –	– –	– –	–

Table 1
The results for Example 1.

Finally, McMillan has a technique to do unbounded model checking using Sat solvers, where Sat-solvers are exploited to manipulate sets of states stored as boolean formulas [18]. We plan to explore whether games can also be solved using a similar approach.

References

[1] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *Proceedings of the Tenth International Conference on Concurrency Theory*, volume 1664 in LNCS, Springer, pp. 82–97, 1999.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.

[3] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Proc. of the 10th Int'l Conf. on Computer-Aided Verification*, vol. 1427 of LNCS, pages 521–525. Springer-Verlag, 1998.

[4] Arnold A. and Niwinski D. Rudiments of μ-calculus. Studies in Logic and the Foundations of Mathematics 146. 2001.

Array size	Bdd method	Bounded methods		
	Mocha	Step(k)	Qbf method	Sat method
			$M1$	Tree
5	0 (5)	5 6 7	3 32 257	3 188 –
6	1 (6)	5 6 7	9 93 872	23 16718 –
7	9 (7)	5 6 7	22 841 3872	81 – –
8	77 (8)	5 6 7	41 1901 10746	1895 – –
9	518 (9)	5 6	– –	11764 –

Table 2
The results for Example 2.

[5] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso *MBP: a Model Based Planner*. In *Proc. of IJCAI-2001 workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[6] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 in LNCS, pages 193–207. Springer-Verlag, 1999.

[7] A. Biere. μcke - Efficient μ-calculus model checking. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, volume 1254 of LNCS, pages 468–471. Springer-Verlag, 1997.

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

[9] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proc. of the 14th Int'l Conf. on Computer-Aided Verification*, LNCS 2404, pp.428–441. Springer-Verlag, 2002.

[10] A. Church. Logic, arithmetics, and automata. In *Proceedings of the International Congress of Mathematicians, 1962*, pages 23–35, Institut Mittag-Leffler, 1963.

[11] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P.N. Ashar. Learning from BDDs in SAT-based bounded model checking. To appear, *Proceedings of the 40th Design Automation Conference (DAC'03)*, 2003.

[12] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of Design Automation and Test in Europe (DATE'02)*, pages 142–149, 2002.

[13] E. Giunchiglia, M. Narizzano and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 364–369, 2001.

[14] R. Jensen and M. Veloso. OBDD-based Universal planning for synchronized agents in non-deterministic domains. Journal of Artificial Intelligence Research, 13, pages 189-226, 2000.

[15] O. Kupferman, and M. Y. Vardi. Module checking. In *Proceeding of the 8th International Conference on Computer-Aided Verification*, volume 1102 of LNCS, pages 75–86. Springer-Verlag, 1996.

[16] Reinhold Lets. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proc. of Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of LNCS, pages 160–175. Springer-Verlag, 2002.

[17] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of LNCS, pages 250–264. Springer-Verlag, 2002.

[19] 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.

[20] J. Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1192–1197, Morgan Kaufmann Publishers, 1999.

[21] J. Rintanen. Constructing conditional plans by a theorem prover. Journal of Artificial Intelligence Research, 10:323–352,1999.

[22] P.J.G. Ramadge, and W.M. Wonham. The control of discrete event systems. Proc. IEEE, 77:81-98, 1989.

[23] Tommi Junttila. Boolean circuit package version 0.20. `http://www.tcs.hut.fi/~tjunttil/circuits/index.html`

[24] W. Thomas. Infinite games and verification. In *Proc. of the 14th Int'l Conf. on Computer-Aided Verification*, vol. 2404 of LNCS, pp.58–64. Springer-Verlag, 2002.

[25] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proc. of Int'l Conference on Computer Aided Design (ICCAD'02)*, 2002.

SAT-Based Methods for Sequential Hardware Equivalence Verification Without Synchronization

Zurab Khasidashvili and Ziyad Hanna

Logic and Validation Technology
Design Technology Division
Intel Development Center, Haifa, Israel
{zurab.khasidashvili, ziyad.hanna}@intel.com

Abstract

The BDD- and SAT-based model checking and verification methods normally require an initial state. Here we are concerned with sequential hardware verification, where an initial state must be one of the reset states. In practice, a reset state is not always given by the designer, and computing a reset state of a circuit is a hard problem. In this paper we propose a method allowing usage of SAT-based verification methods without a need for a user-given or a computed initial state. The idea is to employ a binary encoding of 3-valued modeling of circuits, and use the undefined state X as a reset state.

1 Introduction

In the theory of Finite State Machines (FSM) [Koh78], one assumes an *initial* state (or a set of initial states), from which the machine starts operating. Here we will be concerned with sequential verification of synchronized hardware (circuit) models. In the practice of hardware verification, an initial state s_i of a circuit C is a state where all state elements (latches and flip-flops) have a binary value (T or F), and there is an *initializing sequence* π_i that brings C from the X state to that binary state s_i [CA89]. A *reset* or a *synchronization* sequence for C, on the other hand, is a sequence π_r that brings C from any *binary* state to a unique state s_r, called a *reset* or a *synchronization* state (π_r and s_r are independent from the state from which C starts operating) [Koh78]. Any initializing sequence is clearly a reset sequence, but the converse does not hold [CA89].

Classic BDD-based model checking and verification algorithms require a reset state [CBM89,CM90,TSLBS90,CCQ96,CC97,McM93]. The same is true for well known SAT-based model checking algorithms such as *Bounded Model Checking* [BCC99,BCCFZ99] or the *induction* method [SSS00]. Computation of reset sequences is a hard problem [CJSP93,PB94,PJH94,LP96,KBS96,CPRSS97,RH02].

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

Therefore in this work we are looking for verification methods that can work without a reset state.

Unlike SAT and BDD-based methods, the ATPG methods do not require a reset state. There, one assumes the outputs to differ, and looks for a *justifying assignment*. The circuit modeling is *ternary* – besides the two binary values T and F, one considers an *unknown* value, X (elsewhere also denoted by $\perp$ or u). A justified assignment gives an input vector sequence that, if applied to the circuits starting at the unknown state X (or at *any* binary state), brings them to a state where their outputs differ.

In order to take advantage of the rapidly developing SAT-based verification technology, here we propose a SAT-based method for verifying 3-valued equivalence of sequential circuits without initialization. Our method is based on the *dual-rail* modeling of circuits, where every ternary value is represented with a pair of binary values (see [Bry87,BS94,SB95,KR03]). Via dual-rail encoding, we can arrive to ordinary (2-valued) propositional logic formulation of the verification problem.

The novelty of our approach is to show that the dual-rail X state can be used as a reset state in the (forward as well as backward) SAT-based algorithms mentioned above (the BMC and induction algorithms). We first present an algorithm for checking 3-valued equivalence which uses the X state as a reset state, and prove its correctness and completeness. We then discuss the applicability of our method to verification with respect to other concepts of sequential equivalence, such as *alignability* or *post-synchronization* equivalence [Pix92], and *steady-state* equivalence [KMH02].

The paper is structured as follows. In the next section, we quickly recall some basic definitions used in this work. In Section 3, we recall a backward ATPG based algorithm for verifying 3-valued equivalence and explain its drawbacks. In Section 4, we give a light introduction to a binary encoding, called *dual-rail* encoding, of 3-valued logic into Boolean logic, originally developed for the purpose of efficient symbolic simulation and more direct modeling of circuit operation [BS94]. We also refer to more recent results on usage of the dual-rail encoding in SAT-based sequential verification [KR03]. In section 5, we propose a SAT-based method for 3-valued equivalence verification, and discuss how it relates to the ATPG algorithm mentioned above. In Section 6 we discuss how our method can be extended to steady-state and alignability sequential equivalence verification. Experimental results are discussed in Section 7. Conclusions appear in Section 8.

2 Preliminaries

Without restricting generality, we will assume that any circuit C has exactly one output, o. We denote by C_1 and C_2 our specification and implementation circuits (with outputs o_1 and o_2, respectively), and assume that they have the same set of inputs (dummy inputs can be added, if necessary). We denote by C_{xor} the combined circuit with shared inputs and XORed output $o = o_1 \, XOR \, o_2$. And we denote by C_{xnor} the combined circuit (the *product machine* [HS98]) with shared inputs and

XNORed output $o = o_1\, XNOR\, o_2$.

We consider *ternary* modeling of circuit node values. The values could be one of the *binary* values – T or F, or an *undefined* value – $\perp$ (elsewhere also denoted by X or u). Given a ternary (or binary) input vector sequence π, $n(s, \pi)$ will denote the value of node n of a circuit C after 3-valued simulation of C with π, starting at state s. Similarly, $C(s, \pi)$ denotes the (ternary) state into which π brings C, from state s.

A circuit C is specified as a collection of *next-state functions* (NSFs) of the latches as well as of the output. An NSF is a function of current and next-state values of inputs and latches. [1] This collection of NSFs defines a *sequential instance* corresponding to C, denoted $Inst(C)$. We denote by $Pins(C)$ the set of inputs, latches, and the output of C. Every pin variable p can be viewed as a sequence $(p[m])_{m\geq 0}$ of Boolean variables, each $p[m]$ representing value of pin p at phase m (thus the next state of $p[m]$ is $p[m+1]$).

Assumptions and *proof obligations* can be added to an instance. Assumptions are assumed to be true in all (relevant) time phases, and proof obligations represent properties whose validity in all (relevant) phases we intend to check. The proof obligations we will be interested in are safety properties related to the validity of $o_{xnor} \Leftrightarrow T$.

Unrolling to depth k of the instance $Inst(C)$ yields a *combinational instance*, denoted $C[0, k]$, consisting of variables $\{p[i] \mid 0 \leq i \leq k, p \in Pins(C)\}$, and the relations on them are induced by the NSFs. The function of the output o in $C[0, k]$ at time phase k will be denoted by $o[0, k]$. We assume it is a partial function on all Boolean variables in the instance; partial because some value combinations are illegal as they contradict the NSF relations imposed on the instance. Alternatively, $o[0, k]$ can be seen as the conjunction of all NSF relations and assumptions in $C[0, k]$.

Intuitively, falsification of a proof obligation expressing $o_1[k] \Leftrightarrow o_2[k]$ in $C[0, k]$ corresponds to k iterations of an ATPG procedure of finding a counter-example (CEX) to the proof obligation $o_1 \Leftrightarrow o_2$. We will see in the later sections that this correspondence is not as tight as it may seem from the first sight.

The following example clarifies the above definitions.

Example 2.1 Consider a circuit C that consists of a negated latch l, with data d and clock which is always false: $c = F$ (see Figure 1). Let $o = \neg l$ denote the output. Then $Inst(C)$ consists of two NSFs: $l' = c'\&d' + \neg c'\&l = l$ and $o' = \neg l'$, where x' denotes next state value of x. Unrolling $Inst(C)$ to depth 1 yields combinational instance $C[0, 1]$ consisting of variables $o[0], o[1], l[0], l[1], d[0], d[1]$ and relations $o[i] = \neg l[i]$ for $i = 0, 1$, and $l[1] = l[0]$.

[1] Thus, the circuits that we consider are Mealy machines.

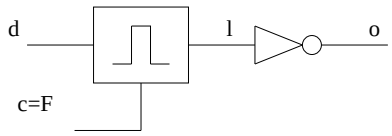

Fig. 1. Negated Latch

3 A backward ATPG based method for verification without initialization

Huang et al [HCC01] developed an ATPG based method for verifying 3-*valued safe replacement* as well as *3-valued equivalence* for sequential circuits with or without an initial state. To define 3-valued equivalence, they introduced a *covering* relation on signals with ternary values: signal v_1 covers signal v_2 iff whenever $v_1 = T$ or $v_1 = F$, then $v_1 = v_2$.

Definition 3.1 • Circuit C_2 with output o_2 is called 3-*valued safe replacement* of circuit C_1 with output o_1 iff for any input sequence π, $o_1(\perp, \pi)$ *covers* $o_2(\perp, \pi)$. (That is, when o_1 has a binary value, then o_2 must have the same binary value.)

- Circuits C_1 and C_2 are *3-valued equivalent*, written $C_1 \cong_3 C_2$, iff for any input sequence π, $o_1(\perp, \pi) = o_2(\perp, \pi)$.

The values $\{(T,T),(F,F),(\perp,\perp)\}$ for the output pair (o_1, o_2) are called 3-*valued equal-pairs* of C_1 and C_2; in this case, C_{xnor} is in 3-*valued equal-state*. The remaining pairs $\{(T,F),(F,T),(\perp,T),(\perp,F)\}, (T,\perp),(F,\perp)\}$ are called 3-*valued differ-pairs*, and C_{xnor} is in 3-*valued differ-state* in this case.

An input vector sequence π such that $(o_1(\perp,\pi), o_2(s_2,\pi)) \in \{(T,F),(F,T)\}$ is called a *partial test* for C_1 and C_2 in [HCC01] (this definition is not symmetric). Note that such a π brings C_{xnor} from state $\perp$ into a 3-valued differ-state. When $s_2 = \perp$, π is also called a *test* sequence (for stack-at-false for o_{xor}).

To check for 3-valued safe replacement, the authors propose to use an ATPG solver in the following way:

The backward justification for the $o_{xor} = T$ (on C_{xor}) stops whenever one of the following two conditions is satisfied:

- *(Unjustifiable condition)*: All state requirements generated during the search of a partial test sequence are proven unjustifiable. Then C_2 is 3-valued safe replacement of C_1.
- *(Justified condition)*: A state requirement that does not have requirements on C_1 is reached. Then a partial test sequence has been found, and C_2 is not a 3-valued safe replacement of C_1.

Similarly, 3-valued equivalence can be disproved by generating a state requirement has no requirements on C_1 or on C_2. And 3-valued equivalence can be proved by showing that all those state requirements that are generated while searching for a partial test for C_1 and C_2 and for a partial test for C_2 and C_1, are unjustifiable.

The above algorithm needs a termination criterion, based on some sort of 'diameter' or a fix-point, to be complete. For example, let both C_1 and C_2 be negated

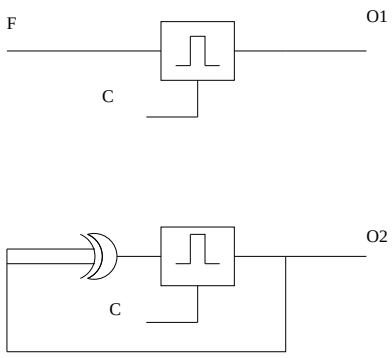

Fig. 2. 3-valued inequivalent circuits.

latches, l_1 and l_2, with control F (like the circuit C in Example 2.1). Then $C_{xor}[0, k]$ will depend on variables $l_1[0]$ and $l_2[0]$ for any k, and neither of the two stopping conditions will ever be satisfied. Thus the algorithm will report 'INDETERMINATE' when a time limit will be reached.

There is also another reason why the above algorithm is not complete: If an input vector sequence that can bring C_{xor} from X state to a differ state (with output T) exists, a partial test for C_{xor} that the backward justification algorithm above is looking for may not exist:

Example 3.2 Consider two circuits C_1 and C_2 (see Figure 2), each consisting of a single latch with clock signal c, with pattern say $c = T, F, T, F, \ldots$. The input of the first latch is constant F, while the input of the other latch is $o_2\, XOR\, o_2$. Starting from the X state, o_1 behaves as $o_1 = X, F, F, \ldots$, and $o2$ behaves as $o_2 = X, X, X, \ldots$. Thus these circuits are not 3-valued equivalent (and C_2 is not 3-valued safe replacement of C_1). However, o_{xor} can never become T if it starts operation from a non-T state (the only two concretizations of the sequence $o_2 = X, X, X, \ldots$ are $o_2 = F, F, F, \ldots$ and $o_2 = T, F, F, \ldots$), thus a partial test doesn't exist for C_1 and C_2.

Remark 3.3 *The above example was pointed out to us as a counter-example to (the sufficiency part of) Lemma 2 of [HCC01], which states that C_2 is a 3-valued safe replacement of C_1 iff there is no partial test for C_1 and C_2. While we believe the above example is not a counter-example to Lemma 2 of [HCC01]* [2] *, the correctness of the lemma does not affect the correctness of the above algorithm or our results below, and we will not elaborate on this issue further (it goes beyond the scope of this paper).*

Note that, intuitively, work with X values in a circuit corresponds to work with QBFs (Quantified Boolean Formulas): latch values are universally quantified in a predicate expressing a stop condition in the ATPG procedure above. Abdulla et al [ABE00] investigated ways to simplify QBF translation into quantifier free

[2] The authors state that they use an enhanced 3-valued logic simulation in Lemma 2; such simulation is based on approximating 3-valued simulation by 2-valued simulation, thus assumes only binary values on inputs as well as initial values of latches (see also Example 4.1). In the algorithm however they use the usual 3-valued simulation.

propositional formulae to facilitate SAT solvers on QBF, for the purpose of SAT-based model checking. Here we pursue a different path: To develop a SAT-based verification algorithm for 3-valued equivalence checking, we consider a dual-rail encoding of the ternary values. In the next section, we give a brief introduction to the subject. We will later explain why this approach can work well with certain SAT solvers, and how it can be extended to verifying sequential equivalence without initialization with respect to other useful concepts of sequential equivalence.

4 Verification using dual-rail modeling of circuits

Dual-rail modeling of circuits was introduced by Bryant [Bry87]. It was used in [BS94] to enable a more precise modeling of circuit operation, and to enable representation of all *ternary* values with BDDs via a binary encoding. It resulted in a more efficient symbolic simulator, as more complex behaviors could be modeled with a single simulation run. We refer to [SB95,Jon02] for more information.

Each ternary value v is encoded as a pair of binary values (v_h, v_l), called the *high* and the *low* values. The undefined value $\perp$ is encoded as a pair $\perp = (T, T)$. The truth constants are encoded by $T = (T, F)$ and $F = (F, T)$. The pair $\top = (F, F)$ encodes a *contaminated* or *over-specified* value. To avoid any confusion, we use F^{dr}, T^{dr}, and $\perp^{dr}$ to denote the dual-rail encoding of T, F and $\perp$, respectively. And $v^{dr} = (v_h, v_l)$ will denote the dual-rail encoding of a ternary variable v.

Sequential logic can be expressed by using Boolean logic connectives such as $\&, +$, and $\neg$, and a *phase-delay* or *next state* operation, $'$. Thus in order to model sequential logic in dual-rail, it is enough to have dual-rail rules for these operations. We overload these logic connectives to denote the corresponding dual-rail counterparts as well. These dual-rail rules are as follows: Let $x^{dr} = (x_h, x_l)$ and $y^{dr} = (y_h, y_l)$ be dual-rail encoding of ternary variables x and y. Then

- $(x_h, x_l)\&(y_h, y_l) = (x_h \& y_h, x_l + y_l)$;
- $(x_h, x_l) + (y_h, y_l) = (x_h + y_h, x_l \& y_l)$;
- $\neg(x_h, x_l) = (x_l, x_h)$;
- $(x_h, x_l)' = (x_h', x_l')$.

Thus a dual-rail NSF is a pair of NSFs of the high and low values. We denote by $C^{dr}[0, k]$ the unrolled, to depth k, dual-rail sequential instance, and denote by $o^{dr}[0, k]$ the value of o in that instance (cf. definition of $o[0, k]$ in $C[0, k]$.

Example 4.1 Let us compute $x^{dr}\, XOR\, x^{dr}$ for $x^{dr} = \perp^{dr}$, as in Example 3.2:

$$\begin{aligned}
\perp^{dr} XOR \perp^{dr} &= ((T,T)\&\neg(T,T))+(\neg(T,T)\&(T,T)) \\
&= ((T,T)\&(T,T))+((T,T)\&(T,T)) \\
&= (T\&T, T\&T)+(T\&T, T\&T) \\
&= (T,T)+(T,T) \\
&= (T+T, T+T) \\
&= \perp^{dr}.
\end{aligned}$$

We can see that dual-rail computation corresponds to usual 3-valued logic. [3]

To ensure that in a sequential instance the inputs are always binary, one needs to add, for any input variable i, an assumption $i_h = \neg i_l$. This in particular will guarantee that we do not introduce (F,F) values in the instance. Further, if (F,F) values are not introduced in assumptions or in proof obligations, the NSFs cannot introduce them either (because the above four operations cannot result in an (F,F) value if the arguments are not over-constrained). Thus, for example, over-constrained values should not appear in a satisfying assignment found by a SAT-solver. An appearance of (F,F) in a satisfying assignment indicates a bug (in the design or in the tool), that is why we don't add to the instance an assumption forbidding over-constrained values on all variables.

We demonstrate dual-rail computation on another example:

Example 4.2 Let C be a circuit as in Example 2.1. Then $Inst(C)$ consists of four NSFs: $l'_h = (c'_h\&d'_h)+(c'_l\&l_h) = l_h$, $l'_l = (c'_l+d'_l)\&(c'_h+l_l) = l_l$, $o'_h = l'_l$, and $o'_l = l'_h$. Besides, we assume that d, as an input, is always binary, by adding $d_h = \neg d_l$ as an assumption to $Inst(C)$.

Dual-rail modeling is currently used in an alignability verification engine, Insight, in the formal verification group at Intel. Despite the double number of variables, experimental results show that the dual-rail implementation is $1.5x$ faster than a single-rail implementation based on the initialization flow reported in [RH02]. Among other factors, this is due to the fact that the dual variables 'behave similarly', and our SAT solver can exploit this similarity without a significant overhead [KR03]. For example, SAT solvers based on the *saturation* method [SS00] are known to perform well when there are many equivalent (up to negation) variables in the instance.

5 A SAT-based method for checking 3-valued equivalence

In this section, we show how the BMC algorithm [BCC99,BCCFZ99] and the induction method [SSS00] can be adapted to enable verification without a reset state, by using the dual-rail state $\perp$ as an initial state. Unlike the original ATPG based

[3] With enhanced 3-valued simulation as in the proof of Lemma 2 of [HCC01], we get $\perp XOR \perp = F$, since $T\,XOR\,T = F\,XOR\,F = F$.

algorithm of Huang et al [HCC01], our algorithm is (sound and) *complete*. We will also see that a more direct encoding of the ATPG algorithm into SAT based dual-rail formalism results in an incomplete algorithm.

Algorithm 1 describes our 3-valued equivalence verification procedure without a reset state.

Algorithm 1 SAT-based algorithm for 3-valued equivalence verification w/o reset state

```
1: Check 3-valued equivalence of o_1 and o_2 {
2: Create a dual-rail sequential instance corresponding to C_xnor ;
3: Bind to T high and low latch values in phase 0 ;
4: Add proof obligation expressing o_xnor^dr ⇔ T^dr ;
5: Apply a complete SAT-based method to the instance ;
6: if a counter-example is generated then
7:    Report 'DIFFER' and EXIT ;
8: end if
9: if the proof obligation is proved then
10:   Report 'EQUAL' and EXIT ;
11: end if
12: else report 'INDETERMINATE' and EXIT ;
13: }
```

Theorem 5.1 *Algorithm 1 is a sound and complete procedure for checking* 3*-valued equivalence.*

Proof. *The situations when the proof obligation can be falsified are exactly the situations where the pair* (o_1, o_2) *is a* 3*-valued differ-pair:*

$$(o_1{}^{dr}, o_2{}^{dr}) \in \{(T^{dr}, F^{dr}), (F^{dr}, T^{dr}), (\perp^{dr}, T^{dr}), (\perp^{dr}, F^{dr}), (T^{dr}, \perp^{dr}), (F^{dr}, \perp^{dr})\}.$$

Thus the algorithm returns 'DIFFER' exactly when $C_1 \not\cong_3 C_2$*, and the counter-example brings* C_{xnor} *from state* $\perp$ *to a* 3*-valued differ state. By the same argument, the algorithm returns 'EQUAL' iff* $C_1 \cong_3 C_2$*. (Only) in case the run terminates without resolving the instance, the algorithm returns 'INDETERMINATE'.* □

In Algorithm 1, we mainly use induction based algorithms [SSS00], since they perform better when a full proof is sought. (We use the BMC based methods in algorithms that require initialization – the counter-examples become (part of) the initializing or synchronizing sequences [RH02,KR03].) We recall briefly that in the induction method, unrolling with increasing depths is performed, till a counter-example (to the proof-obligation) is found, or induction step can be proved (see also [BC00] for a nice description on why a simple induction, with depth 1, is not enough). In [SSS00], termination conditions for induction step are presented that reflect both forward and backward state space traversal methods, thus our algorithm also can be made forward or backward (or combined), depending on which kind of induction is used.

A direct encoding of the ATPG algorithm of [HCC01] into SAT-based model checking problem would correspond to

- Considering the set of (combined) states where all latches of C_1 or all latches of C_2 are in state $\perp^{dr}$ as the set of *initial* states;
- Considering the states where ${o_{xor}}^{dr} = T^{dr}$ as the *bad* states;
- And applying the backward induction scheme of [SSS00].

Counter-examples found by such an algorithm would be the correct ones, but the algorithm would miss counter-examples in situations like in Example 3.2. We therefore abandon this algorithm in favor of Algorithm 1 above.

6 Verification with respect to other concepts of equivalence

In this section, we comment on the applicability of our methods for equivalence checking with respect to some other concepts of equivalence, namely steady-state equivalence and alignability equivalence.

6.1 Verifying steady-state equivalence

We recall definition of steady-state equivalence from [KMH02]. In steady-state equivalence, we compare the outputs only in time phases where both outputs have binary values. Values in other time phases are don't cares. Thus circuits that are 3-valued equivalent are also steady-state equivalent, but not vice versa.

Definition 6.1 ([KMH02])

- An input vector sequence π is called a *steady-state* sequence for a circuit C if $o(\perp, \pi)$ is binary.
- Circuits C_1 and C_2 with outputs o_1 and o_2 are called *steady-state equivalent*, written $C_1 \cong_{ss} C_2$, iff for any input sequence π that is a steady-state sequence for both C_1 and C_2, $o_1(\perp, \pi) = o_2(\perp, \pi)$.

In order to develop a verification procedure for verifying steady-state equivalence without a reset state, we can simply change the proof obligation in Algorithm 1 to the following one: $(binary(o_1)\ \&\ binary(o_2)) \Rightarrow (o_1 \Leftrightarrow o_2)$, where $binary(o_i)$ denotes the property that o_i has a binary value (that is, $o_{ih} = \neg o_{il}$), $i = 1, 2$.

6.2 Verifying alignability equivalence

We recall definition of *alignability* or *post-synchronization* equivalence from [Pix92].

Definition 6.2 • State (s_1, s_2) of the combined circuit C_{xnor} is an *equivalent* state if for any input sequence π, $o_1(s_1, \pi) = o_2(s_2, \pi)$. [4]

[4] The concepts of equal- and differ-states should not be mixed with equivalent and inequivalent states. In this definition, all states are binary.

- A binary input sequence π is an *aligning* sequence for a combined state (s_1, s_2) of C_{xnor} if it brings C_{xnor} from state (s_1, s_2) into an equivalent state.
- Circuits C_1 and C_2 are *alignable*, written $C_1 \cong_{aln} C_2$, if every state of C_{xnor} has an aligning sequence (or equivalently, there is a sequence, called a *universal aligning sequence*, that aligns any state of C_{xnor}).

Lemma 6.3 (i) *If circuits C_1 and C_2 are synchronizable and $C_1 \cong_{ss} C_2$, then $C_1 \cong_{aln} C_2$.*

(ii) *If $C_1 \cong_{aln} C_2$, then $C_1 \cong_{ss} C_2$.*

Proof.

(i) *Let C_1 and C_2 be steady-state equivalent. Consider sequence π that synchronizes both C_1 and C_2, say into a state pair (s_1, s_2). Then for any sequence π', the concatenation of π and π' is a steady-state sequence, thus π' ends in a state (s_1', s_2') where o_1 and o_2 have equal binary values. Thus (s_1, s_2) is an equivalent state pair, implying that $C_1 \cong_{aln} C_2$.*

(ii) *Now let C_1 and C_2 be alignable. Suppose on the contrary that C_1 and C_2 are not steady-state equivalent. Then there is a steady-state sequence π that brings any state into a differ state (with outputs different binary values). Such a sequence can distinguish any pair of states, thus C_1 and C_2 do not have an equivalent state pair, and they cannot be alignable – a contradiction.*

□

Alignability equivalence is a widely used concept of equivalence. Therefore, to show the importance of our methods, it is important to clarify the relevance of our methods for alignability equivalence verification. Indeed, there are a number of ways allowing to infer alignability or non-alignability of circuits by using the methods of checking steady-state or 3-valued equivalence presented in the early sections. We mention a few of them, based on the above lemma and a result in [HCC01].

- If our steady-state verification algorithm proves circuits C_1 and C_2 inequivalent, then it returns a counter-example π_d that brings C_{xnor} from state $\perp$ to *binary* differ-state. Such a sequence π_d is actually a universal counter-example demonstrating that $C_1 \not\cong_{aln} C_2$ (as it can distinguish any pair of states of C_1 and C_2).
- If on the other hand $C_1 \cong_{ss} C_2$, then from the SAT procedure proving this, it is possible to extract information whether the part $binary(o_1)\&binary(o_2)$ becomes true in some phase. Such a procedure depends on the particular strategy used to resolve the sequential instance, and goes beyond the scope of this paper. (Of course initializability of C_1 and C_2 can be checked separately.) If yes, we have actually proven $C_1 \cong_{aln} C_2$ as well. If not, we cannot claim $C_1 \not\cong_{aln} C_2$, as synchronizing but not initializing sequence may exist that brings C_{xnor} into an equivalent state. For such not 3-*valued initializable circuits* [HCC01] we use a formal initialization method, briefly discussed in [RH02], to find an aligning sequence when it exists.
- It is shown in [HCC01] that if both C_1 and C_2 are initializable, then $C_1 \cong_3 C_2$ implies $C_1 \cong_{aln} C_2$. Actually, it is enough to show that one of the circuits is

Ckt	#L	#G	Steady-state equivalence			Alignability equivalence		
			Pass	Probl.	Time (sec.)	Pass	Probl.	Time (sec.)
C_1	712	7838	9	0	1067	9	0	1106
C_2	1208	38259	0	1	1331	0	1	1728
C_3	100	1202	28	0	1128	28	0	2701
C_4	826	6260	7	1	2697	6	2	3921
C_5	154	1730	35	0	2008	35	0	3251
Total			**79**	**2**	**12707**	**78**	**3**	**8231**

Table 1
Comparison of performance (#L latches, #G gates).

initializable and the other one is its 3-valued safe replacement [HCC01].

- Since 3-valued equivalence requires o_1 and o_2 to match in all time phases, the above sufficient condition may not be practical to infer alignability from 3-valued equivalence. Instead, a *($k-$) delayed* 3-valued equivalence can be used, which requires o_1 and o_2 to match from phase k onward. Still, usage of delayed 3-valued equivalence in proving alignability is limited.

7 Experimental results

We have implemented Algorithm 1 and its modified version for checking 3-valued and steady-state equivalences. Most of our circuits are resetable, thus in practice this algorithms performs alignability check as well.

Experiments reported below were performed on 550MHz dual CPU Linux machine with 2GB memory. A timeout of 300 seconds was used in the SAT solver. Experimental results show that say the steady-state equivalence algorithm is $1.5x$ faster than a dual-rail alignability equivalence algorithm that first performs synchronization of the specification and implementation circuits (see Table 1; there, numbers of latches and gates represent an average per output). And as already mentioned, the latter in turn is $1.5x$ faster than a corresponding single-rail implementation of alignability checking engine (despite the fact that dual-rail modeling requires twice as much NSFs) [KR03]. Furthermore, the counter-examples returned by the steady-state engine are in average $2x$ shorter than those found by the alignability engine, which is much more important (for debugging) than the above reported speed-up (see Table 2, where circuits $C_1 - C_7$ contain loops, while circuits $C_8 - C_{14}$ are loop-free; all data is given per single outputs).

Steady-state equivalence								Alignability equivalence			
Ckt	#L	#G	len	Ckt	#L	#G	len	Ckt	len	Ckt	len
C_1	526	2509	9	C_8	151	1747	4	C_1	21	C_8	8
C_2	18	160	6	C_9	173	2037	6	C_2	8	C_9	8
C_3	18	92	6	C_{10}	107	1263	6	C_3	11	C_{10}	12
C_4	18	415	4	C_{11}	112	1317	6	C_4	5	C_{11}	9
C_5	24	207	4	C_{12}	67	744	4	C_5	6	C_{12}	9
C_6	25	1121	8	C_{13}	57	619	4	C_6	10	C_{13}	10
C_7	704	7660	11	C_{14}	98	726	3	C_7	65	C_{14}	6
Total							**81**				**188**

Table 2
Comparison of counter-example length (#L latches, #G gates, len = CEX length).

8 Conclusions

Thus far, SAT-based verification methods have been mainly concentrated on property checking, and for the good reason: It is well understood that circuit equivalence verification can be performed by the model-checking of properties that express equivalence of the circuit outputs. Indeed, in this work, we have demonstrated how SAT-based methods (such as the BMC or the induction method) can be used for proving sequential equivalence in accordance with a number of important sequential equivalence concepts.

In particular, we have developed SAT-based verification methods for verification of sequential circuits with respect to 3-valued, steady-state and (partly) alignability equivalence. The novelty of our approach is that it does not require a reset state. Instead, we can use the undefined state as a reset state, after encoding the latter into a binary representation. Unlike the ATPG-based method of [HCC01], from which our approach emerged, our algorithms for checking 3-valued and steady-state equivalence are complete. We hope that our work sheds further light on the relationship between the ATPG- and SAT-based sequential verification.

An advantage of our approach is that the verification procedure becomes relatively simple conceptually, thus it is easy to implement and maintain it. Our method *compliments* earlier methods for which synchronization is an essential part of verification, as our algorithms outperform (in a number of dimensions) similar algorithms that need to compute reset states. Clearly, this does not decrease the importance of initialization based methods. In particular, synchronization methods when initializing sequences do not exist are indispensable.

Actually, because of the importance of short counter-examples for debugging at early stages of design, steady-state verification is entering a default flow in our

verification methodology, which was previously based on initialization. We remark also that the ability to find counter-examples quickly is important in the framework of model abstraction refinement (see e.g. [CGJLV00]). There, because one works with pruned models, there is a higher probability of (false) negatives, till a right pruning is found. And synchronization can be checked on correctly pruned models only, when the pruned models are steady-state equivalent.

Despite the rapid development and success of SAT-based model checking, there is still a long way to go. As an example, we mention that, on loop-free circuits, SAT-based equivalence methods (both with or without initialization) perform very poorly compared to the method developed in [KMH02] for loop-free circuits. Both steady-state and alignability checks time out after thousands of seconds on tests that can be verified in less than a minute with the method in [KMH02]. SAT-based model checking will profit from the development of alternative ways of translating model-checking problems into SAT problems.

Acknowledgments We thank R. Fraer, A. Jas, D. Kaiss, J. Moondanos, A. Rosenmann and G. Wolfovitz for careful reading, and Shi-Yu Huang for clarifying the subtleties of his ATPG method for checking 3-valued equivalence.

References

[ABE00] P. A. Abdulla, P. Bjesse, N. Eén. *Symbolic reachability analysis based on SAT solvers*, Tools and Algorithms for the Construction and Analysis of Systems, TACAS'00, Springer LNCS, 2000.

[BC00] P. Bjesse, K. Claessen. *SAT based verification without state space traversal*, FMCAD'00, Springer LNCS, 2000.

[BCC99] A. Biere, A. Cimatti, E. Clarke. *Symbolic model checking without BDDs*, Tools and Algorithms for the Construction and Analysis of Systems, 1999.

[BCCFZ99] A. Biere, A. Cimatti, E. Clarke, M. Fujita. Y. Zhu. *Symbolic model checking using SAT procedures instead of BDDs*, DAC 1999.

[Bry87] R. E. Bryant. *Boolean Analysis of MOS Circuits*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-6, No. 4, 1987

[BS94] R.E. Bryant, C.-J.H. Seger. *Digital circuit verification using partially-ordered state models*, Twenty-Fourth International Symposium on Multiple-Valued Logic, 1994.

[BCLMD94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L. Dill. *Symbolic model checking for sequential circuit verification*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol 13, n. 4, 1994.

[CCQ96] G. Cabodi, P. Camurati, S. Quer. *Improved reachability analysis of large finite state machines*, ICCAD 1996.

[CC97] G. Cabodi, P. Camurati. *Symbolic FSM traversals based on the transition relation,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, n. 5, 1997.

[CA89] K.-T. Cheng, D. Agrawal. *State assignment for initializable synthesis,* ICCAD'89.

[CJSP93] H. Cho, S.-W. Jeong, F. Somenzi, C. Pixley. *Synchronizing sequences and symbolic traversal techniques in test generation,* J. Electron. Test.: Theory & Applications, vol. 4, n. 2, 1993.

[CGJLV00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-guided Abstraction Refinement,* CAV'00, Springer LNCS, 2000.

[CGP99] E. Clarke, O. Grumberg, D. Peled. *Model Checking,* MIT Press, 1999.

[CPRSS97] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda and G. Squillero. *A new approach for initialization sequences computation for synchronous sequential circuits,* IEEE VLSI in Computers and Processors, 1997.

[CBM89] O. Coudert, C. Berthet, J.C. Madre. *Verification of synchronous sequential machines based on symbolic execution,* Workshop of Automatic Verification Methods for Finite State Systems, 1989.

[CM90] O. Coudert, J.C. Madre. *A Unified framework for the formal verification of sequential circuits.*, ICCAD 1990.

[HS98] G.D. Hachtel, F. Somenzi. *Logic Synthesis and Verification Algorithms,* Kluwer Academic Publishers, 1998.

[HCC01] S.-Y. Huang, K.-T. Cheng, K.-C. Chen. *Verifying sequential equivalence using ATPG techniques,* ACM Transactions on Design Automation of Electronic Systems, 2001.

[Jon02] R.B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification,* Kluwer Academic Publishers, 2002.

[KR03] D. Kaiss, A. Rosenmann. *Dual rail modeling for SAT based sequential verification.* (In preparation.)

[KBS96] M. Keim, B. Becker and B. Stenner. *On the (non)-resetability of synchronous sequential circuits,* IEEE VLSI Test Symposium, 1996.

[KMH02] Z. Khasidashvili, J. Moondanos, Z. Hanna. *TRANS: Efficient Sequential Verification of Loop-Free Circuits.* HLDVT'02, IEEE Computer Society Press, 2002.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory,* McGrawHill, New York, 1978 (second edition).

[LP96] Y. Lu and I. Pomeranz. *Synchronization of large sequential circuits by partial reset,* IEEE VLSI Test Symposium, 1996.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem,* Kluwer Academic Publishers, 1993.

[Pix92] C. Pixley. *A theory and implementation of sequential hardware equivalence*, IEEE transactions on Computer-Aided Design, vol. 11, no. 12, 1992.

[PB94] C. Pixley, G. Beihl. *Calculating resetability and reset sequences*, ICCAD, 1991.

[PJH94] C . Pixley, S.-W. Jeong, G.D. Hachtel. *Exact calculation of synchronizing sequences based on binary decision diagrams*, IEEE transactions on Computer-Aided Design, vol. 13, 1994

[PR96] I. Pomeranz, S.M. Reddy. *On removing redundancies from synchronous sequential circuits with synchronizing sequences*, IEEE transactions of computers, vol. 45, no. 1, 1996.

[RH02] A. Rosenmann, Z. Hanna. *Alignability equivalence of synchronous sequential circuits*, IEEE International High Level Design Validation and Test Workshop, HLDVT'02, IEEE Computer Society Press, 2002.

[SB95] C.-J. H. Seger, and R. E. Bryant. *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design, vol. 6, no. 2, 1995.

[SS00] M. Sheeran, G. Stålmarck. *A tutorial on Stålmarck's method of propositional proof*. Formal Methods In System Design, 16 (1), 2000.

[SSS00] M. Sheeran, S. Singh, G. Stålmarck. *Checking safety properties using induction and a SAT-solver*, FMCAD, 2000.

[TSLBS90] H. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli. *Implicit enumeration of finite state machines using BDDs*, CAD'90, 1990.

A Satisfiability-Based Approach to Abstraction Refinement in Model Checking [1]

Bing Li [2] Chao Wang [3] Fabio Somenzi [4]

University of Colorado at Boulder

Abstract

We present an abstraction refinement algorithm for model checking of safety properties that relies exclusively on a SAT solver for checking the abstract model, testing abstract counterexamples on the concrete model, and refinement. Model checking of the abstractions is based on bounded model checking extended with checks for the existence of simple paths that help in deciding passing properties. All minimum-length spurious counterexamples are eliminated in one refinement step by a procedure that combines the analysis of the conflict dependency graph produced by the SAT solver while looking for concrete counterexamples with an effective abstraction minimization procedure.

1 Introduction

Model checking [CGP99] is an algorithmic approach to the verification of properties of reactive systems, which has been successfully applied to both hardware and software. Since model checking entails the exploration of a potentially very large state space, the alleviation of the so-called state explosion problem has been the object of much research. On the one hand, techniques have been developed that allow models with hundreds of state variables to be analyzed directly. On the other hand, abstraction has been used to allow the model checker to draw conclusions on the original, concrete model by examining a simpler, abstract one.

For systems with many state variables and many transitions, the symbolic approach has proved crucial. In symbolic model checking, sets of states and transition are described by their characteristic functions. Various forms of representation have been used for these functions, the most popular being Binary Decision Diagrams (BDDs) [Bry86], and Conjunctive Normal Form (CNF).

[1] This work was supported in part by SRC contract 2001-TJ-920 and NSF grant CCR-99-71195.
[2] Email: Bing.Li@Colorado.EDU
[3] Email: wangc@Colorado.EDU
[4] Email: Fabio.Li@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

Classical BDD-based model checking [McM94] is based on the computation of fixpoints. For instance, the reachable states of a model are computed as the least fixpoint of the function $\lambda Z \,.\, I \vee \mathrm{Succ}(Z)$, which adds the successors of the states in Z to the initial states. Both the set of states and the successor relation are stored as BDDs. The fixpoint computation converges in a number of iterations that equals the maximum distance of a reachable state from the initial states. Checking for convergence is made easy by the strong canonicity of BDDs (identical sets share the same representation). BDD-based model checking can therefore prove properties almost as easily as it can disprove them.

Bounded Model Checking (BMC) [BCCZ99], on the other hand, formulates the reachability test as a series of satisfiability (SAT) checks for paths of bounded length. (To see if a path of length k to a set of states exists, the transition relation is unrolled k times.) For finite systems the process must eventually terminate: the length of the shortest path between two states cannot exceed the number of states. Hence, if no path is found with length up to the number of states, the target states are known to be unreachable. This observation, however, does not help for the kind of models that one encounters in practice. The diameter of the state graph would give a much better bound on k, but, unfortunately, it is hard to compute [BCCZ99]. For this reason, BMC has come to be regarded as an excellent *debugging* (as opposed to *verification*) technique. That is, classical BMC is particularly adept at finding counterexamples, but ill-suited to prove their absence.

The ability demonstrated by BMC to deal with models beyond the reach of BDD-based methods has sparked interest in the use of CNF and SAT for proof as well as refutation. Two main approaches have been pursued: The replacement of BDDs with CNF formulae in the fixpoint computation [ABE00,WBCG00,McM02], and the development of more effective termination criteria for BMC.

The opportunity of replacing BDDs with CNF formulae can be argued on the grounds that canonicity of representation makes BDDs somewhat inflexible. Hence, some functions that admit compact representations in CNF have exceedingly large BDDs. However, the inflexibility argument can also be used against CNF, and memoization techniques are more effective for BDDs. In fact, to date, CNF-based fixpoint computation has not demonstrated a consistent advantage over the classical BDD-based one. One may argue that the main reason for the success of BMC in finding counterexamples lies in its avoidance of the needless computation and storage of reachable states that are not on the error trace.

Several proposals have been made to improve BMC's ability to prove the non-existence of a path. It is straightforward to check for inductive invariants, since it only entails checking for the existence of a transition from a state that satisfies the invariant to one that does not. An extension of the inductive approach has been presented in [SSS00], in which termination occurs as soon as the length of the path reaches the length of the longest simple path from an initial state, or to a target state. A recent paper [McM03] proposes the analysis of the unsatisfiable formulae to allow termination when the *reverse sequential depth* of the model is reached.

Early termination in BMC requires additional checks beyond the one for the

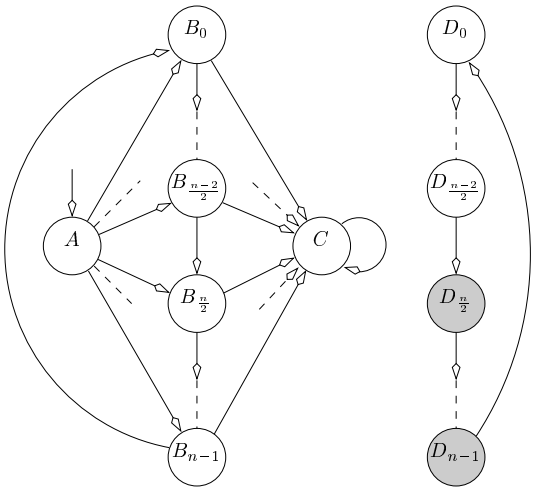

Fig. 1. Model with long simple path

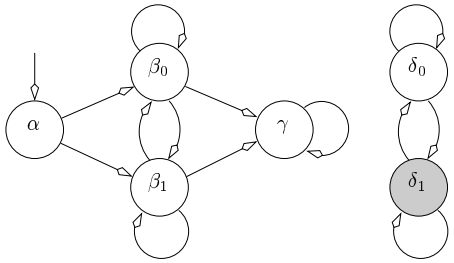

Fig. 2. Abstraction of the model of Fig. 1

existence of paths of certain lengths. These checks translate into more clauses in the CNF formulae whose satisfiability has to be established. For the approach of [SSS00], the number of extra clauses is quadratic in the length of the path. As a result, it is not surprising that finding counterexamples is slower than with pure BMC. The extra cost, however, appears to be worth paying, since it increases substantially the fraction of passing properties that can be decided. Unfortunately, there remain instances for which the additional termination tests are too expensive. Consider the model illustrated in Fig. 1. It has $2n+2$ states, one of which is initial (A). The $n/2$ states $D_{n/2}, \ldots, D_{n-1}$ are the (unreachable) target states. The longest simple path from the initial state has length $n+1$, while the longest simple path to a target state that does not visit any other target state has length $n/2$; the reverse sequential depth of the model is also $n/2$. Hence, the methods of [SSS00,McM03] will have to consider paths of length $n/2$ before they can declare the target states unreachable. By contrast, the forward sequential depth is 2.

Fig. 2 shows an abstraction of the model of Fig. 1. States A, B_i, C, and D_i are abstracted by α, $\beta_{\lfloor 2i/n \rfloor}$, γ, and $\delta_{\lfloor 2i/n \rfloor}$, respectively. The target state remains unreachable in this model, and the forward sequential depth is still 2; however, the longest simple path and the sequential depth are reduced. Though in general there is no guarantee that abstraction will shorten or even not lengthen the longest simple paths, or the shortest paths, this example illustrates how abstraction may help BMC,

especially for passing properties.

Abstraction and BMC have been combined in more than one recent work, especially in the context of abstraction refinement. In abstraction refinement [Kur94], one starts with a coarse abstraction of the given, concrete model and keeps refining it until the property is decided. For universal properties like the reachability properties that are the focus of this paper, this often means that the abstract models simulate the concrete one [Mil71], and that either the property is shown to hold on an abstract model, or a counterexample is found in the concrete one. In [WHL+01,CGKS02,CCK+02,WLJ+03] BMC is used to check whether counterexamples found in the abstract models can be *concretized*, that is, whether a counterexample can be found in the concrete model that is mapped by the abstraction onto the abstract counterexample. The first three of these methods also analyze the failed concretization test to guide the refinement. Therefore, they represent instances of counterexample-guided abstraction refinement. On the other hand, [WLJ+03] analyzes the abstract model to decide how to refine it. Yet another approach is the one of [MA03], in which the abstract model is derived from a failing BMC run on the concrete model. This reversal of the customary order is attractive for those frequent cases in which paths of moderate length can be easily checked on the concrete model.

One common trait of the approaches to abstraction refinement mentioned so far is the application of a BDD-based model checker to the abstract models, and of SAT solvers to the concrete ones. By contrast, the objective of this paper is to explore what can be achieved with a SAT solver as the only decision procedure in the abstraction refinement framework. The rationale for combining BDDs and SAT is that each is well-suited to the task assigned to it: The SAT solver is good at checking the existence of a path of a given length in a large model, whereas the BDD-based model checker is better at proving the absence of certain paths, regardless of their lengths, in a model of moderate size. This observation is certainly well motivated when one regards the models for which abstraction refinement results have been reported in the literature; their sizes rarely exceed 1,000 binary state variables. As the models grow larger, however, we expect an approach purely based on SAT to become more competitive. Therefore, our goal is to eventually being able to switch between BDD-based model checking and SAT-based techniques for the analysis of the concrete model. In this paper we report on a significant step in that direction by presenting an algorithm for abstraction refinement that is purely based on SAT.

Our approach is similar to the ones discussed so far in the fact that abstractions are obtained by removing part of the state variables of the model; refinement then consists of reinstating some of the removed variables. The algorithm has three major components: the decision procedure for the abstract model is the one of [SSS00], which has already been mentioned. The second component—the choice of the refinement—combines elements of [WLJ+03] and [CCK+02]. Like the former, it addresses all the abstract counterexamples at once; like the latter, it analyzes the conflict dependency graph of the failed concretization test to derive a set of candidate state variables from which the ones that will be added to the abstract model

are chosen. Finally, the third component is a heuristic procedure for abstraction minimization. This minimization is quite important in our approach, because the simultaneous elimination of all spurious counterexamples of a certain length tends to generate large sets of candidate variables. Our experimental evaluation of the SAT-based abstraction refinement algorithm compared it to both BMC (with and without early termination checks for passing properties) and to the best abstraction refinement algorithm available to us [WLJ+03]. The results, discussed in Section 4, show that the new approach, though not uniformly superior, is more robust than the others, and is especially promising for the more challenging problems.

2 Preliminaries

Let $V = \{v_1, \ldots, v_n\}$ be a set. 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 $\{v^i_1, \ldots, v^i_n\}$. We define an *open system* as a 4-tuple

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

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.

We assume that the transition relation is given as the composition of elementary relations. If $W = \{w_1, \ldots, w_m\}$ with $m \geq n$, our assumption amounts to writing:

$$T(V, W, V') = \bigwedge_{1 \leq i \leq n} (v'_i \leftrightarrow w_i) \wedge \bigwedge_{1 \leq i \leq m} T_i(W, V) \ . \tag{1}$$

We consider the case of a sequential circuit, in which the variables in W are associated with the primary inputs and the outputs of the combinational logic gates of the circuit; the variables in V are associated with the memory elements. Each T_i is called a *gate relation* because it usually describes the behavior of a logic gate. For instance, if w_i is the output variable of a two-input AND gate with inputs w_j and v_k, then $T_i = w_i \leftrightarrow (w_j \wedge v_k)$. If, on the other hand, w_i is a primary input to the circuit, then $T_i = 1$. Each term of the form $v'_i \leftrightarrow w_i$ equates a next state variable to a combinational variable. (The output of the gate feeding the i-th memory element.)

In a sequential circuit, a state variable v_j is said to be in the *direct support* of variable v_i (w_i) if the memory element associated to v_j is connected to the memory element (logic gate) associated to v_i (w_i) by a path that goes through logic gates only. Variable v_i is in the *cone of influence* (COI) of v_i (w_i) if there is a path (of any kind) connecting v_j to v_i (w_i).

An open system Ω defines a labeled transition structure in the usual way, with states Q_Ω corresponding to the valuations of the variables in V, and transition labels corresponding to the valuations of the variables in W. Conversely, a set of states $S \subseteq Q_\Omega$ corresponds to a predicate $S(V)$ or $S(V')$. Predicate $S(V)$ ($S(V')$) is the

characteristic function of S expressed in terms of the current (next) state variables. State $q \in Q_\Omega$ is an initial state if it satisfies $I(V)$. State set $S \subseteq Q_\Omega$ is *reachable* from state set S' in k steps if there is a path of length k in the labeled transition structure defined by Ω that connects some state in S' to some state in S; equivalently if

$$S'(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, W^i, V^i) \wedge S(V^k) \tag{2}$$

is satisfiable. State set S is reachable from S' if there exists $k \in \mathbb{N}$ such that S is reachable in k steps from S'. A state set is reachable (in k steps) if it is reachable (in k steps) from I. When no confusion arises we shall identify a state $q \in Q_\Omega$ with the set $\{q\}$. A finite (infinite) sequence of states $\rho \in Q_\Omega^*$ ($\in Q_\Omega^\omega$) is a finite (infinite) *run* of Ω if the first state is initial, and every other state is reachable from its predecessor in one step. The set of all possible runs of Ω is the language of Ω, denoted by $L(\Omega)$.

A linear-time *safety property* P of Ω is a subset of Q_Ω^ω such that any infinite sequence over Q_Ω not in P has a finite prefix that cannot be extended to a sequence in P [AS85]. Open system Ω satisfies safety property P if $L(\Omega) \subseteq P$. Checking the satisfaction of an ω-regular safety property P by an open system Ω can be reduced to the reachability problem by composing Ω with an automaton $\mathcal{A}_P$ that accepts the inextensible prefixes of the sequences not in P. The property is satisfied by the open system if no state of the composition $\Omega \parallel \mathcal{A}_P$ that projects on an accepting state of $\mathcal{A}_P$ is reachable. In the sequel we restrict ourselves to ω-regular safety properties, and assume that the given open system already incorporates the property automaton. This assumption allows us to identify the property with a set of (accepting) states of the system, which we also denote by P. Hence, property P is satisfied by Ω if there is no $k \in \mathbb{N}$ such that

$$I(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, W^i, V^i) \wedge \neg P(V^k) \tag{3}$$

is satisfiable. An invariant is a safety property that states that a certain predicate holds of all reachable states of Ω. In this case P is the set of states that satisfy that predicate.

The search for a k such that (3) is satisfiable can obviously be restricted to the range $\{0, \ldots, |Q_\Omega| - 1\}$. Hence, in theory, the process is guaranteed to terminate. In practice, the number of states is too large to be of any practical use, and tighter upper bounds for k are sought. In model checking approaches that are based on fixpoint computations [McM94,ABE00,WBCG00,McM02], the maximum value of k is provided by the number of iterations needed to reach convergence. On the other hand, for algorithms that directly check the satisfiability of (3), the diameter of the graph [BCCZ99] or bounds obtained from the structure of the hardware model have been proposed [BKA02]. Here we summarize a method proposed in [SSS00] that is of particular interest to us.

A *simple path* is one that visits a state at most once. If some state in $\neg P$ is reachable, there must exist a simple path from an initial state to it that does not go

through any other states in I or $\neg P$. Hence, if 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 $\neg P$ and no other state is in $\neg P$, then, there is no path of length greater than or equal to k connecting a state in I to a state in $\neg P$. If in addition, there is no path of length less than k connecting I to $\neg P$, then $\Omega \models P$. Two sets of states S' and S are connected by a simple path of length k in Ω if

$$\Sigma_k(S', S) = S'(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, W^i, V^i) \wedge S(V^k) \wedge \bigwedge_{0 \le j < i \le k} \bigvee_{1 \le l \le n} (v_l^i \neq v_l^j) \quad (4)$$

is satisfiable. Checking the two conditions above then amounts to checking that either of the following formulae is unsatisfiable.

$$\Sigma_k(I, Q) \wedge \bigwedge_{0 < i \le k} \neg I(V^i) \quad (5)$$

$$\Sigma_k(Q, \neg P) \wedge \bigwedge_{0 \le i < k} P(V^i) \ . \quad (6)$$

Note that the predicate corresponding to the set Q is true.

Abstract interpretation [CC77] provides a very flexible framework for the description of abstraction. In this paper, however, we consider the following restricted definition. Open system $\widehat{\Omega} = \langle \widehat{V}, \widehat{W}, \widehat{I}, \widehat{T} \rangle$ is an *abstraction* of Ω if

- $\widehat{V} \subseteq V$;
- $\widehat{W} \subseteq W$ such that $v_i \in \widehat{V}$ implies $w_i \in \widehat{W}$;
- $\widehat{I}(\widehat{V}) = \exists (V \setminus \widehat{V}) \, . \, I(V)$;
- $\widehat{T}(\widehat{V}, \widehat{W}, \widehat{V}') = \exists (V \setminus \widehat{V}) \, . \, \exists (W \setminus \widehat{W}) \, . \, \exists (V' \setminus \widehat{V}') \, . \, T(V, W, V')$.

(Note that w_i is the combinational variable associated to v_i'.) Property $\widehat{P}$ is the abstraction of property P with respect to $\widehat{\Omega}$ if $\widehat{P}(\widehat{V}) = \exists (V \setminus \widehat{V}) \, . \, P(V)$. If P is an ω-regular set and $\widehat{\Omega}$ satisfies (or models) $\widehat{P}$, then Ω satisfies P. That is,

$$\widehat{\Omega} \models \widehat{P} \rightarrow \Omega \models P \ . \quad (7)$$

This preservation result is the basis for the following abstraction refinement approach to the verification of P. One starts with a coarse abstraction $\widehat{\Omega}_0$ of the *concrete* open system Ω and checks whether $\widehat{\Omega}_0 \models \widehat{P}_0$. If that is the case, then $\Omega \models P$; otherwise, there exists a least $k' \in \mathbb{N}$ such that

$$\widehat{I}(\widehat{V}^0) \wedge \bigwedge_{1 \le i \le k'} T(\widehat{V}^{i-1}, \widehat{W}^i, \widehat{V}^i) \wedge \neg \widehat{P}(\widehat{V}^{k'}) \quad (8)$$

is satisfiable. The satisfying assignments to (8) are the shortest-length *abstract counterexamples* (ACEs). If $\widehat{\Omega}_0 \not\models \widehat{P}_0$ one or more ACEs are checked for *concretization*. That is, one checks whether (3) has solutions that agree with the ACE(s)

being checked. Because of the additional constraints provided by the ACEs, a concretization test is often less expensive that the satisfiability check of (3). However, its failure only indicates that the abstract error traces are spurious. Therefore, if the concretization test fails, one chooses a refined abstraction $\widehat{\Omega}_1$ and repeats the process, until one of these cases occurs.

(i) $\widehat{\Omega}_i \models \widehat{P}_i$ for some i, in which case $\Omega \models P$ is inferred.

(ii) The concretization test passes for some i, in which case it is concluded that $\Omega \not\models P$ and the satisfying assignment found is returned as counterexample to P.

(iii) The refinement eventually produces $\widehat{\Omega}_i = \Omega$. In this final case, the satisfiability check of (8) answers the model checking question conclusively. This is an undesirable outcome because the purpose of abstraction is defeated.

When the refinement $\widehat{\Omega}_{i+1}$ of $\widehat{\Omega}_i$ is chosen with the help of the information provided by the failed concretization test, one talks of counterexample-guided abstraction refinement.

The cone of influence (direct support) of a property is the union of the cones of influence (direct supports) of all the variables mentioned in the property. Cone-of-influence reduction refers to the abstraction in which $\widehat{V}$ is the COI of the property. It is commonly applied before any model checking is attempted, because it satisfies

$$\widehat{\Omega} \models \widehat{P} \leftrightarrow \Omega \models P \ . \tag{9}$$

3 Algorithm

Our algorithm is shown in Fig. 3. Initially, an abstract model $\widehat{\Omega}$ is computed by collecting only the state variables (called *latches* henceforth) in the direct support of the property P. The algorithm then progressively increases L from its initial value 0 until either a counterexample of length L is found in the concrete system Ω, or it is concluded that no counterexample exists in the current abstract model. If at some point, the abstract model becomes the concrete model, the endgame is executed as described in Lines 14–19.

Lines 3–13 verify the abstract models. First, (5) and (6) are checked to see whether the simple path conditions are met. If either one is unsatisfiable, the property holds, and the algorithm terminates. Otherwise, the algorithm checks whether there is a counterexample of length L in the abstract model, by checking (3) on $\widehat{\Omega}$; if there is no length-L abstract counterexample, there is no counterexample of length up to L in the concrete model either. (This is because every abstract model simulates the concrete model; hence, if there is a real counterexample of length $L' \leq L$ in the concrete model, there must be a corresponding abstract counterexample of length $L'' \leq L'$. Since the counterexample length is increased in increments of one, we would have found this counterexample before.) Since there is no counterexample of length up to L (in either the abstract model or the concrete model), L is increased by one. On the other hand, if there is an abstract counterexamples of

```
boolean pureSAT(Ω,P) {
1   L = 0;
2   Ω̂ = CreateInitialAbstraction(Ω,P);
3   while (Ω̂ ≠ Ω) {
4       if (¬ CheckSimplePath(Ω̂,P,L))
5           return TRUE;
6       if (ExistCex(Ω̂,P,L)) {
7           if (ExistCex(Ω,P,L))
8               return FALSE;
9           refinement = GetRefinementFromCA(Ω,Ω̂,P,L);
10          Ω̂ = AddRefinementToAbsModel(Ω̂, refinement);
11      }
12      L = L + 1;
13  }
14  while (CheckSimplePath(Ω,P,L)){
15      if (ExistCex(Ω,P,L))
16          return FALSE;
17      L = L + 1;
18  }
19  return TRUE;
}
```

Fig. 3. The PureSAT algorithm

```
set GetRefinementFromCA(Ω, Ω̂,P,L) {
    nsVarSet = GetNextStateVarsFromCDG(Ω,P,L);
    sufficient = ∅ ;
    while (sufficient does not kill all length-L counterexamples
            ∧ nsVarSet is not empty) {
        someNsVars = PickVarsThreshold(nsVarSet, threshold);
        sufficient = sufficient ∪ someNsVars
        nsVarSet = nsVarSet \ someNsVars
    }
    RCArray = ComputeRelativeCorrelationArray(sufficient,Ω,Ω̂);
    return RefinementMinimization(Ω̂,RCArray);
}
```

Fig. 4. The refinement algorithm

length L, (3) is checked on the concrete model to see if any concrete counterexample of the same length exists. If it does, the property fails; otherwise, the refinement step (Lines 9–10) is executed.

The goal of the refinement procedure is to find a minimal set of latches not in $\widehat{\Omega}$ which, after being added to the abstract model, can kill all the counterexamples

of the length L. Our refinement algorithm is based on computing and analyzing the *unsatisfiable core* [GN03,ZM03] associated with the proof that there is no concrete counterexample of length L; hence, it is similar to the conflict analysis method proposed in [CCK+02]. However, our approach differs significantly from [CCK+02] in the following aspects:

(i) The authors of [CCK+02] first identify a single spurious abstract counterexample (by using BDD-based model checking), together with its failure index. (I.e., the time step from which the ACE is no longer concretizable in the concrete model.) A conflict dependency graph is built from the unsatisfiable BMC obtained by constraining the concrete model with the single spurious ACE up to the failure index time step. The refinement set is then computed by analyzing the conflict dependency graph. In our algorithm, however, we do not use a single abstract counterexample to constrain the BMC instance (and we do not compute the failure index). Rather, an unconstrained BMC instance (on the concrete model, for path length up to L) is used for the concretization test; such a BMC instance covers all the possible length-L spurious abstract counterexamples.

(ii) In [CCK+02], the *invisible* latches (those not currently in $\widehat{\Omega}$) are added to the refinement set if their corresponding literals at the failure index time step appear in the conflict dependency graph. In our algorithm, all the literals (which correspond to either latches or internal logic gates at different time steps) appearing in the unsatisfiable core are recorded in the SAT solver. However, only those invisible latches whose *next-state variable* literals (i.e., the literals corresponding to the input variable of a latch at a different time step) appear in the unsatisfiable core are added to the refinement set. This refinement set, when added to $\widehat{\Omega}$, is sufficient to kill all length-L spurious abstract counterexamples. Our algorithm for picking refinement variables is shown in Fig. 4. The original "sufficient set" (i.e., nsVarSet in the pseudo code) may or may not be minimal; hence, refinement minimization is used to get rid of the redundant latches in the refinement set before the function returns. In some cases, the number of redundant invisible latches in nsVarSet may be too large, causing REFINEMENTMINIMIZATION to spend too much time. The **while** loop, together with a threshold, is used to heuristically get a smaller "sufficient set" for the refinement minimization: Each time, only a certain number of invisible latches are picked from nsVarSet, after which (3) is checked to see if they are already sufficient.

(iii) Our refinement minimization algorithm is also somewhat different from [CCK+02]. Both methods remove redundant latches greedily. Each latch in turn is tentatively removed. If (3) remains unsatisfiable, the remaining latches are still sufficient, and the dropped latch is indeed redundant; otherwise, that latch is restored to the refinement set. In our method, the order in which invisible latches are removed in the minimization procedure is based on the *relative correlation* of each candidate latch to the current abstract model. The relative

correlation of an invisible latch equals the ratio of the number of gates in the COI of this latch which are already in the abstract model divided by the total number of gates in the COI of this latch. Intuitively, the larger the relative correlation of a latch, the larger effect it will have when added to or subtracted from the current abstract model. The invisible latches of the current sufficient set are sorted by Function COMPUTERELATIVECORRELATIONARRAY: The one with the smaller relative correlation is considered of less importance, and thus will be tested for deletion earlier. In this way, we can concentrate on the important invisible latches and at the same time keep the refined abstract model small.

Our approach is also related to the one of [MA03]. Both approaches check all counterexamples of a certain lenght at once by a model checking run on the concrete model. The main differences are:

(i) We use SAT, instead of a BDD-based model checker, for the abstract model. This will give our method an advantage in proofs that require an abstract model of size comparable to that of the concrete one.

(ii) Our abstraction grows at each refinement, and we use refinement minimization to control its size, whereas the abstraction of [MA03] is computed from scratch each time. Refinement minimization requires repeated BMC runs; these, however, are runs on the abstract model. In the experiments reported in Section 4, refinement minimization was never the bottleneck, and it could be further sped up by using an incremental SAT solver.

4 Experimental Results

To evaluate the technique of Section 3, we compared four algorithms: an implementation of the BMC [BCCZ99] algorithm, BMC extended with the checks for simple paths [SSS00] (referred to as SSS), our PURESAT algorithm, and the GRAB algorithm of [WLJ+03], which uses both BDDs and SAT. All the four algorithms are implemented in VIS-2.0 [B+96,VIS], and Chaff [MMZ+01] was use as the back-end SAT solver. The experiments were run under Linux on an IBM IntelliStation with a 1.7 GHz Intel Pentium 4 CPU and 2 GB of RAM.

The comparison was conducted on 26 models, either from industry or from VIS verification benchmarks [B+96,VIS] except for *lsp*. This model was created to illustrate the help BMC could get from abstraction. A simplified version of it appears in Fig. 1. Since in the concrete model, the longest simple path is long, SSS failed to complete, even though PURESAT finished within one second.

The results are shown in Table 1. The first column is the name of the model, the second column indicates whether each property passes or fails; if a property fails, the number in this column is the length of the counterexample. The third column gives the number of latches in the cone of influence of the property. The fourth column lists the time of BMC. A time in parentheses is the time elapsed when the process ran out of memory. In our experiments, the time limit was set to 8 hours.

Table 1
Experimental results. Boldface is used to highlight best CPU times

model	pass/ cex length	latches in COI	BMC time	SSS time	PureSAT		Grab	
					time	final sz.	time	final sz.
lsp-p1	pass	12	>8h	>8h	**1**	3	**1**	3
D12-p1	16	48	**5**	25	37	23	14	23
D23-p1	5	85	**1**	**1**	3	25	20	21
D2-p1	14	94	**6**	25	20	48	180	48
D14-p1	14	96	**65**	83	1460	80	>8h	(75)
D1-p1	9	101	**1**	5	11	20	9	21
D1-p2	13	101	**2**	12	26	23	51	23
D1-p3	15	101	**3**	18	32	23	56	25
I12-p1	370	119	>8h	>8h	>8h	(12)	**2503**	16
B-p1	pass	124	>8h	>8h	2074	18	**173**	18
B-p2	17	124	150	675	247	7	**93**	7
B-p3	pass	124	>8h	>8h	>8h	(42)	**223**	43
B-p4	pass	124	>8h	(23708)	>8h	(43)	**393**	42
D22-p1	10	140	**2**	10	17	132	720	132
D24-p1	9	147	7	10	2	4	**1**	4
D24-p2	pass	147	>8h	16	6	8	**3**	8
D24-p3	pass	147	>8h	**1**	4	6	20	8
D24-p4	pass	147	>8h	**1**	4	6	43	8
D24-p5	pass	147	>8h	**1**	4	8	3	5
M0-p1	pass	221	>8h	(2537)	2156	13	**136**	16
D5-p1	31	319	58	592	155	13	**31**	18
D18-p1	23	506	**96**	795	4359	160	>8h	(99)
D16-p1	8	531	**10**	29	31	14	92	14
D20-p1	14	562	**26**	101	6228	232	>8h	(69)
rcu-p1	pass	2453	>8h	(3115)	**136**	11	195	10
IU-p2	pass	4493	(11331)	>8h	**1756**	14	>8h	(6)

The fifth column is the time of SSS; the sixth column shows the time for PURESAT; the seventh column is the number of latches in the final abstract model. If the time is greater than 8 hours, the number in parentheses in the next column is the number of latches in the abstract model when time ran out. The next two columns are the data for GRAB. All CPU times are in seconds except when noted.

The algorithm labeled BMC can check inductive invariants. However, no such properties are included in our set of experiments. From the table we can see that, in general, for passing properties, PURESAT is better than both BMC and SSS. For failing properties, with a few exceptions, BMC is best, while PURESAT is better than GRAB. For the largest model, like IU, whose COI contains 4493 latches, PURESAT is the only one being able to verify the property. Interestingly, GRAB and PURESAT fail to finish similar numbers of experiments (4 for GRAB and 3 for PURESAT). However, the two sets of failures are disjoint. This is an encouraging sign for the development of a hybrid algorithm that may switch between BDDs and SAT for the analysis of the abstract models.

Though PURESAT appears to be reasonably robust, there are only three cases in Table 1 in which it manages to be fastest. This is in part due to the fact that the implementation is still preliminary.

5 Conclusions

We have presented an abstraction refinement algorithm for model checking safety properties that uses a SAT solver as sole decision procedure. We have compared this algorithm to both BMC and to an abstraction refinement algorithm that uses both BDDs and CNF SAT. The new algorithm is competitive and was the only one to complete the largest test case. Our implementation is still preliminary. We plan to investigate the use of an incremental SAT solver like SATIRE [WKS01] in the abstraction minimization phase, which is currently the most time consuming part of the algorithm. We are also interested in the extension of the techniques of [WLJ+03] to the SAT environment. This is not an entirely trivial task, since they are based on the knowledge of the sets of states at various distances along the paths connecting initial states to error states.

By its very nature, the PURESAT algorithm suffers, albeit in attenuated form, from the same problems that afflict the basic procedure used in analyzing the abstract models. Improvements like those proposed in [McM03] may boost PURESAT's performance. More generally, the integration with a BDD-based approach to the analysis of the abstract model should lead to a more robust and powerful approach to abstraction refinement.

References

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for the Construction of Systems (TACAS)*, pages 411–425, 2000. LNCS 1785.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[B+96] 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.

[BCCZ99] 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, March 1999. LNCS 1579.

[BKA02] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 151–165. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977.

[CCK+02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, November 2002. LNCS 2517.

[CGKS02] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*, pages 886–891, Munich, Germany, March 2003.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[MA03] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, April 2003. LNCS 2619.

[McM94] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[McM02] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Fifteenth Conference on Computer Aided Verification (CAV'03)*. Springer-Verlag, Berlin, July 2003. LNCS 2725. To appear.

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[MMZ+01] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[SSS00] 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, November 2000. LNCS 1954.

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

[WBCG00] P. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 124–138. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[WHL+01] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.

[WKS01] 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.

[WLJ+03] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. Submitted for publication, April 2003.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Munich, Germany, March 2003.

A BMC–Formulation for the Scheduling Problem in Highly Constrained Hardware Systems

Gianpiero Cabodi Sergio Nocco Stefano Quer

Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

Alex Kondratiev Luciano Lavagno Yosinori Watanabe

Cadence Design Systems, Inc.
Berkeley, CA

Abstract

This paper describes a novel application for SAT–based Bounded Model Checking (BMC) within hardware scheduling problems.

First of all, it introduces a new model for control-dependent systems. In this model, alternative executions (producing "tree-like" scheduling traces) are managed as concurrent systems, where alternative behaviors are followed in parallel. This enables standard BMC techniques, producing solutions made up of single paths connecting initial and terminal states.

Secondly, it discusses the main problem arising from the above choice, i.e., re-writing resource bounds, so that they take into account the artificial concurrencies introduced for controlled behaviors.

Thirdly, we exploit SAT-based Bounded Model Checking as a verification technique mostly oriented to bug hunting and counter-example extraction. In order to consider resource constraints, the solutions of modifying the SAT solver or adding extra clauses are both taken into consideration.

Preliminary experimental results, comparing our SAT based approach to state-of-the art BDD-based techniques are eventually presented.

1 Introduction

Synthesis of efficient and high performance control units and data paths from high-level behavioral specifications has long been considered a very promising technique for tackling the ever growing complexity of digital design. At the same time, it is a very elusive goal, because after more than twenty years of intensive research, and even the appearance on the market of some indus-

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

trial CAD tools, high-level synthesis is still far from being widely used as its predecessors, register-transfer level and logic synthesis.

Within this framework, BDD-based manipulations have recently attained interesting results, as an alternative to ILP and heuristic techniques. In this approach a non-deterministic finite automata describes design alternatives for highly-constrained control-dominated models. After that, the automata's state space is symbolically visited, adopting model checking's state-of-the-art techniques. These techniques are mix of forward and backward traversals, aimed at finding a scheduling solution as a trace connecting initial and terminal states.

In the simplest case of systems without control choices (*if-then-else* construct), a schedule is a path, and symbolic scheduling works just like invariant checking with counter-example extraction. However, control-dependent behavior produces scheduling instances as DAGs (or trees), where *fork* and *join* nodes are introduced to represent scheduling choices, depending on values of control operands. This has required a specific backward traversal procedure (called *validation* in [3]), which, albeit not far from standard BDD-based traversals, is not directly mapped to standard Model Checking (e.g., CTL) procedures.

In this work we propose to change the original automaton model introduced in [2,3,4,11] for control-dependent systems, so that standard model checking procedures are supported. More specifically we transform alternative subtraces to concurrent behaviors which are followed in parallel. In this way the resulting scheduling is always a path (instead of a DAG) connecting initial and final states. As a byproduct, we can exploit SAT-based Bounded Model Checking. Indeed, as the designer's aim is to *find* a schedule, not to prove its absence, we believe BMC can work at its best, as a verification technique mostly oriented to bug hunting and counter-example extraction, rather than proof of correctness. Nevertheless, in order to enable this method, we also must re-write the resource bounds, so that they take into account the artificial concurrencies introduced for controlled behaviors.

As a final remark, notice that many High Level Synthesis tools use Control Data Flow Graphs (CDFGs) as their internal model and do not model well constraints coming from input/output operations with the external world (e.g., synchronization, min/max rate, jitter, etc.) and often mostly data dependencies are handled, while control is either ignored or handled by complete case splitting [1]. Although we use CDFGs as the input specification for our tool, we adopt the model introduced by [3], which is at the same time *formal* (based on concurrent automata), *efficient* (it is possible to use symbolic representation techniques with enhancements derived from concurrent specification models), *control-oriented* (condition evaluation and speculative execution are

[1] Approaches that specifically address control-intensive CDFGs (such as [8]) have been introduced only recently.

specific features of [3]), and *flexible* (I/O constraints can be represented by restrictions on the automata state space). As [3] we represent implicitly the full solution space by means of the state space of a product of automata.

2 Background

We assume the reader is familiar with BDDs, SAT and Bounded Model Checking. As a consequence we briefly review only the basic concepts within our application framework.

2.1 High-Level Synthesis Methodologies

Historically two basic approaches have been used for scheduling: Heuristics algorithms and Integer Linear Programming. On the one hand, priority-based heuristic methods (e.g., [10]) can accommodate a variety of data-dominated and control-dominated behaviors, quickly finding good solutions for large problems. On the other hand, they may fail to find an optimal solution in tightly constrained problems, where early pruning decisions may exclude candidates eventually leading to superior solutions. Integer Linear Programming methods (e.g., [7]) can solve scheduling exactly. However, the ILP complexity significantly increases by considering control constraints (if-then-else and loops), and thus may lead to unacceptable execution times. Moreover, they consider only one solution at a time, and hence are not particularly suitable for interactive synthesis.

2.2 Symbolic Scheduling

More recently [2,3,4,11] symbolic methods have been proved effective in finding exact solutions in highly constrained problem formulations.

In [11], the authors propose a symbolic formulation that allows speculative operation execution and exact resource-constrained scheduling. In [2,3], the authors improved the previous method by proposing a new efficient encoding to reduce execution time. This encoding only indicates "whether or not" and not "when" an operation has been scheduled. Finally, [4] handles loops in Data Flow Graphs (DFGs).

Their scheduling technique (as well as ours) assumes an input in the form of a CDFG. A CDFG is a directed acyclic graph describing both data-flow and control dependencies between the operations. Operation nodes are atomic actions potentially requiring the use of hardware resources for one or more clock cycles. Directed arcs establish a link between each operation and the predecessors that produce data required by it. A source and a sink are added before every operation without predecessors and after every operation without successors. Conditional behavior is specified by means of fork and join nodes, and directed arcs also establish a link between the operation evaluating the condition and the related fork/join pair. Operations that are neither connected

by a directed path, nor mutually exclusive due to a preceding fork node, are concurrent [2].

Example 2.1 Figure 1 shows an example of CDFG. In particular Figure 1(a) shows the pseudo-code for a conditional statement and Figure 1(b) the corresponding CDFG.

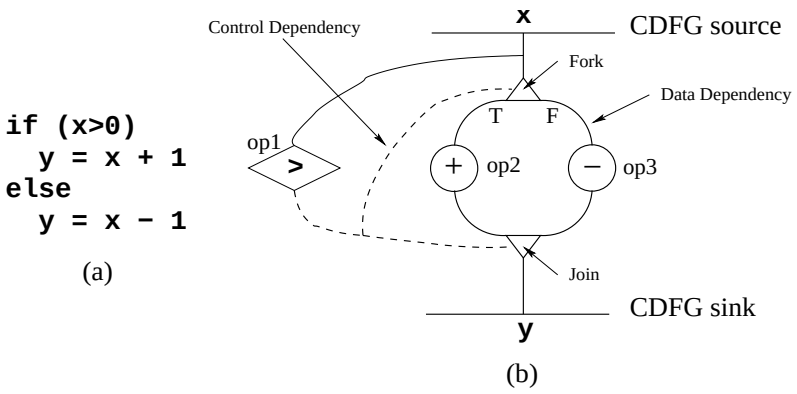

Fig. 1. An example of CDFG.

2.3 Scheduling Automata

A scheduling problem, originally described as a CDFG, can be translated into an automaton, defined by the four-tuple $(V, \mathsf{TR}, S_i, S_f)$, where V is the finite, non-empty set of states, $\mathsf{TR} : V \rightarrow V'$ is the transition relation, and S_i and S_f are respectively the sets of initial and final states.

The generic i-th operation in the CDFG (excluding fork and join operations) is modeled by a two-state automaton. Its transition relation is encoded with exactly two Boolean variables (p_i for the present state and n_i for the next state), with the following meaning:

- $p_i = 0, n_i = 0$: operation i has not been scheduled previously and will not be scheduled in the next cycle.
- $p_i = 0, n_i = 1$: operation i has not been scheduled previously and will be scheduled in the next cycle.
- $p_i = 1, n_i = 0$: operation i has been scheduled previously but the result will no longer be available in the next cycle; this is forbidden in [3], as well as in our solution, in order to reduce the amount of equivalent schedules generated.
- $p_i = 1, n_i = 1$: operation i has been scheduled previously and the result remains available.

The complete scheduling is the Cartesian product of the above automata restricted by several constraints, each one representing a particular allowed be-

[2] The same model, if the sink is connected back to the source, can also be viewed as a safe Petri Net. In this paper we use the automata-based notation for consistency with [11].

havior.

$$\mathsf{TR}(p,n) = \prod_i (\overline{p_i} + n_i) \cdot \mathsf{TR}_{dep}(p,n) \cdot \mathsf{TR}_{res}(p,n)$$

The modeling automaton described by TR encapsulates all legal execution sequences of a system. Fundamentally, it represents multiple legal execution sequences via nondeterministic choices, yet a real implementation must make deterministic choices. If nondeterministic choices are pruned to leave only one deterministic choice, or if multiple choices are made deterministic by conditions, then a finite state machine controller may be directly synthesized. The criterion used to eliminate nondeterminism is usually minimum execution latency. Variations of this exist for control-dependent behavior, where some control cases might be more favored than others.

Let us briefly summarize here dependency and resource constraints, since they will be used in the sequel:

- TR_{dep} represents data dependencies, i.e., it is illegal to schedule an operation with a predecessor that has not yet been scheduled:

$$\overline{p_i} n_j \text{ is illegal for all } i \rightarrow j \text{ data dependencies (dd)}$$

$$\mathsf{TR}_{dep}(p,n) = \prod_{i \rightarrow j \in \mathsf{dd}} (p_i + \overline{n_j})$$

- TR_{res} represents resource constraints. Let us have a resource set with b resources of a given kind (e.g., multipliers) available, and a set ρ of operations competing for such a resource set. It is illegal to schedule more than b concurrent operations from ρ.

$$(\overline{p_i} n_i \cdot \ldots \cdot \overline{p_k} n_k) \text{ with } \{i..k\} \in \rho \text{ is illegal if } |\{i..k\}| > b$$

Let $S_0(p)$ be the initial state of the scheduling product automaton, in which no operation has been scheduled. The set of states reachable at the i-th clock cycle may be computed by a standard iterative image computation:

$$S_i(n) = Img(\mathsf{TR}, S_{i-1}) = \exists_p [\mathsf{TR}(p,n) \cdot S_{i-1}(p)] \tag{1}$$

Valid schedules are represented by state paths that reach a final set of states in which terminal operations have been scheduled.

The exploration techniques presented here are directed by a minimum latency objective. They determine whether, given all constraints imposed and a target latency l, a valid execution sequence of length $\leq l$ exists. With control-dependent models, some additional validity criteria are imposed, and speculative execution may allow some operations after a fork and before a join to be scheduled before the condition evaluation has been scheduled.

3 Handling control dependence through concurrency

Unlike the simpler case of data flow graphs, a witness schedule for control-dependent models is not a single path in a path set but rather a set of paths from start to final states. Such a set of paths is called an *ensemble schedule* in [3] and must include a path for each distinct control-dependent execution sequence. For instance, a RISC processor must be able to execute all instructions and therefore an ensemble schedule for a RISC processor contains sequences for every instruction. As a more specific example, consider some control-dependent behavior that branches into two sets of behaviors depending on a true/false control resolution. An ensemble schedule for this example must contain a path from the start state to the final one that represents execution of true control resolution behavior, and another path that covers false control resolution behavior.

In BDD-based formulation, this requires the introduction of control guard variables, representing non-deterministic choices of each controlling operation. A guard is a binary abstraction of the data value controlling a branching condition. A "completeness" check (i.e., all guard values have reached the terminal state) is added to termination conditions.

Furthermore, a validation procedure operates a backward pruning over the state sets computed by forward BFS. Validation is the most expensive symbolic operation and the main cause for BDD blow-up. It consists of a preimage routine with universal quantification of control guards at control resolution points. This is necessary to enforce causality (identical initial subpath) for outgoing paths at fork points.

Apart from complexity issues, branching schedules and the related validation steps are a major problem for a SAT-based formulation. In order to avoid them, we interpret choice vertices as concurrent forks, and we transform alternative branches into concurrent paths.

So we remove fork and join nodes from the CDFG, and we replace them with unconditioned data dependencies. As a result, a CDFG becomes a DFG, and SAT can explore simultaneously all conditional branches of the original CDFG.

Figure 2 shows the above transformation applied to the example of Figure 1.

Fork and join have been removed, control dependency maintained (as data dependency) for the operations *following the join recombination*. Therefore, in our solution joins work as synchronization points, as no operation following a join is allowed to be executed if both the branches of the control resolution have not been completed yet. This means that our model does not allow control prioritization (as we always have the worst delay), but we have no loss if the objective is minimizing the worst case execution latency. Moreover, since we remove the dependency at forks, speculation is still allowed.

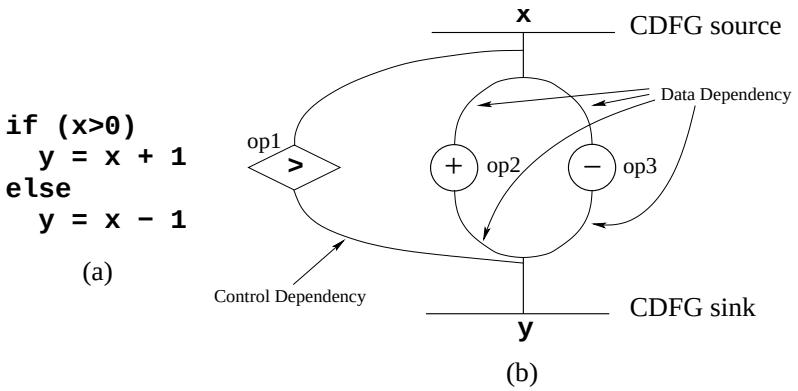

Fig. 2. A CDFG after fork/join removal.

4 Concurrent forks and resource constraints

The artificial fork concurrencies we have introduced have no side-effect in the case of scheduling with unbounded resources. In fact, given any set of concurrent operations, if a "large enough" set of resources can be allocated, all operations may always be executed.

The case of bounded resources is less trivial. In this case not all concurrent executions are "real" concurrencies, since some of them are just artificial. As a consequence, not all concurrent operations are competing for resources. As a direct outcome, we need to modify resource constraints, to take into account that some operations could be allocated to the same resource at the same execution time.

More specifically, let us work on a CDFG with a set of N operations $Op = \{op_1, ..., op_N\}$, each one mapped to (i.e., executable by a resource of) a resource class within the set $R = \{r_1, ..., r_M\}$. The generic resource class r_i is characterized by a bound b_{r_i}, representing the amount of operation unit available for that class, whereas n_{r_i} is the total number of operations in Op mapped to the r_i resource class. The resource bound problem is obviously trivial for class r_i if $n_{r_i} \leq b_{r_i}$, since there can never be a request of resources greater than the available ones (as for the case of infinite resources).

A much more challenging problem is the case of resource bounds actually reducing the amount of possible concurrencies. Let op_i and op_j be two operations mapped to the same resource class, scheduled for concurrent execution (there is a state transition where $\overline{p_i} n_i \overline{p_j} n_j$ holds). Then, resource allocation may fall in one of the following three cases:

- Unconditioned concurrency. The two operations do not belong to different conditional branches in the original CDFG, so their concurrency is a "real" one, requiring the allocation of two resources.
- Mutual exclusion. The two operations are controlled by mutually exclusive conditions, i.e., they are on different branches of some fork in the original CDFG. Their concurrence is artificial, so just one resource is required.

- Speculative execution. Speculation occurs whenever an operation is executed before its controlling condition is resolved. If op_i and op_j are both executed before their distinguishing condition is known, concurrence is real, and two resources are required.

In other words, we may have couple of operations for which concurrency might be unconditioned (first item in the above list), and other ones characterized by conditioned concurrency (second and third item).

4.1 Resource bounds within the SAT solver

Resource bounds can be accounted for directly by a SAT solver. In this solution the SAT solver has to be properly modified in order to count the allocated resources while recursively building a scheduling solution. This is a special purpose solution to follow only in the case the generated overhead is negligible. It basically relies on identifying active operations through variable decisions and implications, and keeping resource allocation counters. A resource conflict occurs whenever an allocation counter is greater than the allowed bound.

4.2 Resource bounds as a Boolean constraint

Although the above solution is feasible, we prefer exploring an alternative one, that is compatible with a generic SAT solver, since no modification to the SAT algorithm is necessary. We simply follow BDD-based approaches, by generating a resource constraint for the transition relation (TR_{res}), which filters out invalid sets of concurrent executions.

There are various strategies for building such a constraint as a Boolean function returning true on allowed sets of operation executions.

4.2.1 Cliques of concurrency graph

A straightforward approach works on the graph of possible concurrencies, where operations are nodes and edges connect pairwise concurrent operations. Such a graph can be generated as the transitive closure of a graph where pairs of operations are concurrent if no data dependency connects them and no resolved control makes them mutually exclusive. The graph can be viewed as an upper bound of concurrencies within a schedule. Given the projection of the concurrency graph to resource class r_i, cliques of size larger than the allowed bound (b_{r_i}) are forbidden.

This is an attractive solution, especially for explicit enumeration, but it is practically limited to small cases, due to its binomial complexity. In particular, it blows up in problems with high degree of concurrency, such as, for example, models of pipelined behaviors.

4.2.2 R-combination filtering function

A more efficient formulation, for the case of unconditioned concurrency, is proposed in [2] for BDD representation. If we omit considering data dependencies, and we simply work on operations of the same resource class, then the resource bound constraint is an r-combination expression, selecting combinations of up to b_{r_i} operations out of n_{r_i}. We call this filtering function $\mathsf{Rfilter}(Op, bound)$. Its size complexity, when expressed as a BDD instead of a two level form, is $O(n_{r_i} \cdot b_{r_i})$, i.e., number of operations mapped to the class times the bound for the class. The function is easily translated to CNF format (with intermediate additional variables), with similar complexity.

Unfortunately, as previously shown, we have conditioned (i.e., artificial) concurrencies, that complicate our model compared to [2], and make the above solution exponential in the number of control choices (forks): we should expand one instance of concurrency graph for each case of resolved/unresolved control operation.

4.2.3 Hybrid two-level approach

Since none of the two previous approaches alone is able to efficiently solve our problem, we developed a hybrid technique, which follows the concurrency graph strategy locally, within control components of the CDFG, and the r-combination approach on a global perspective.

More in detail, we express the resource constraint function (for a given resource class r_i) as a composition of two sub-functions

$$\mathsf{TR}^{r_i}_{res}(p, n) = \mathsf{Rfilter}(\mathsf{Alloc}(p, n), b_{r_i})$$

The outermost function is (a slight modification of) the previously described r-combination filter, whereas Alloc is a function that remaps operation transitions to a set of allocation variables, with the following rules:

- Each uncontrolled operation op_i is remapped to an allocation variable $a_i = \overline{p_i} n_i$, which evaluates true when the operation is executing.
- Controlled subsets of the CDFG (subgraphs included between fork and join nodes) are globally remapped to a proper set of allocation variables, over whom the Alloc functions returns a number of ones exactly corresponding with the amount of resources required. So all artificial concurrencies and/or speculations are taken into account by this function.

The composition is never computed explicitly, but intermediate allocation variables are kept and transferred to the CNF formulation of TR_{res}, which allows us to face the main size bottlenecks: (1) The complexity of conditional concurrency (function Alloc) is kept within small regions of the CDFG. Especially for the important case of looping and/or pipelined behaviors, modeled by serial and parallel instances of the same reference CDFG, this makes the size of Alloc linear in the number of serial/parallel instances. (2) Rfilter, the

function taking care of the overall problem, has size $O(n_{r_i} \cdot b_{r_i})$, i.e., it is linear in the number of operations, for a given resource bound. As an overall result, our result constraint function is scalable, and well suited for looping and pipelined behaviors, which are the most difficult problems in BDD-based approaches.

4.2.4 Implementation details

Figure 3 and 4 show the pseudo-code of the Alloc and Rfilter functions respectively. For sake of simplicity it is assumed that all operations in the CDFG are mapped onto the same resource class, for which maxAlloc units are available.

In our implementation, every operation is labeled with two attributes: (1) the set of all possibly concurrent nodes and (2) a BDD representing the control case for which the operation is enabled. Actually, in order to cover speculation, the meaning of such a BDD is that the operation is *disabled* if the evaluation of the BDD for the already resolved controls returns 0.

As regards the Alloc function, all possible cliques (over the set of operations belonging to the received sub-graph) are recursively built by means of the auxiliary generateCliques function. At each level of recursion, a new node is added to the previously generated clique, checking for speculative execution. In fact, the AND between the node's enable and the clique's enable returns a 0 result only if the current node and at least one node already belonging to the clique are in two different control branches. Therefore, the concurrency of the node w.r.t. the clique is real only if the controlling operations discriminating the branch are not resolved yet (i.e., the nodes in the new clique are executed speculatively). Such controlling operations are therefore added to the *unresolved* set. The transition corresponding to the new clique is then stored as a BDD, and the cliques of bigger sizes are built (the set of possibly concurrent nodes being restricted as the clique has to be completely connected). Eventually, the last loop in the Alloc function defines the allocation variables: variable a_i takes a value of 1 iff there is a transition in the current sub-graph involving the usage of *at least* i resources of the current resource class.

Once all the resource cliques have been generated, the Rfilter function symbolically builds all valid transitions in terms of the allocation variables. To do this, it combines the allocation variables coming from the different calls to the Alloc function to form an expression representing all possible *illegal* allocations (i.e., those requiring at least maxAlloc+1 resources). Then the complementation of such expression, which indeed represents all allocations of at most maxAlloc resources, is returned (and then directly used as a component of TR).

4.2.5 A small example

Let us consider again the CDFG shown in Figure 1 and let us assume that all the operations are mapped on a single ALU. The CDFG is divided by the algorithm into two sub-graphs: the first is composed by the comparison only,

```
ALLOC (graph)
  for (i ← 1 TO maxAlloc + 1)
    tList[i] ← BDD_ZERO
  for (node ∈ graph.nodesSet)
    cliqueSet ← ∅
    unresolved ← ∅
    enable ← BDD_ONE
    generateCliques(tList, node, cliqueSet, unresolved, enable, node.concur)
  for (i ← 1 TO maxAlloc + 1)
    graph.a_i ← new_var
    TR ← BDD_AND(TR, BDD_XNOR(graph.a_i, tList[i]))

GENERATECLIQUES (tList, node, cliqueSet, unresolved, enable, concurSet)
  if |cliqueSet| > maxAlloc
    return
  newEnable ← BDD_AND_EXIST(enable, node.enable, unresolved)
  newUnresolved ← unresolved
  if BDD_IS_ZERO(newEnable)
    newUnresolved ← unresolved ∪ conflictingControls(enable, node.enable)
    newEnable ← BDD_AND(BDD_EXIST(enable, newUnresolved),
                        BDD_EXIST(node.enable, newUnresolved))
  newClique ← cliqueSet ∪ node
  tList[|newClique|] ← BDD_OR(tList[|newClique|],
                              transition(newClique, newUnresolved))
  newConcur ← concurSet ∩ node.concur
  for (op ∈ newConcur)
    generateCliques(tList, op, newClique, newUnresolved, newEnable, newConcur)
```

Fig. 3. The Alloc function.

```
RFILTER ()
  allocations[0] ← BDD_ONE
  for (k ← 1 TO maxAlloc + 1)
    allocations[k] ← BDD_ZERO
  for (i ← 1 TO N_graphs)
    newAllocations ← allocations
    for (j ← 1 TO maxAlloc + 1)
      for (k ← 0 TO maxAlloc + 1)
        if j + k > maxAlloc + 1
          break
        alloc ← BDD_AND(allocations[k], graph_i.a_j)
        newAllocations[j + k] ← BDD_OR(newAllocations[j + k], alloc)
    allocations ← newAllocations
  return BDD_NOT(allocations[maxAlloc + 1])
```

Fig. 4. The Rfilter function.

whereas the second includes both the ADD and SUBTRACT operations. Then the relations defined by the two calls of the Alloc function are respectively:

$$a_1^C = \overline{p_C} n_C, \quad a_2^C = 0$$

and

$$a_1^{AS} = \overline{p_A} n_A + \overline{p_S} n_S, \quad a_2^{AS} = \overline{p_A} n_A \overline{p_S} n_S \overline{p_C}$$

Indeed the first sub-graph may present only a transition requiring 1 ALU unit, whereas the second sub-graph might require 2 ALU units, but only in the case that the ADD and SUBTRACT operations are both executed before the control resolution has been solved. Eventually, the final constraint built by the Rfilter function is:

$$constraint = \overline{a_2^C + a_1^C a_1^{AS} + a_2^{AS}}$$

5 BMC Formulation

Once we have generated the transition relation of the CDFG, as previously described, we have to produce the verification problem which will give us the scheduling solution. This is done by unrolling the transition relation a certain number of times and then trying to prove the mutual reachability between initial and final states.

The BDD representing the transition relation (in monolithic or conjunctive form) is stored as a CNF formula as described in [13].

The verification strategy usually starts with a path of length equal to 1 and increases it till the problem is solved or computation resources are exceeded. For the above reasons the technique works well in falsification and partial verification, whereas full verification is usually achieved by BMC with longer and longer bounds.

Our problem is somehow simpler as, with a proper number of registers, there is always a solution to the scheduling problem. Moreover, our experience shows that unsatisfiable problems are much harder to solve than satisfiable instances. To this respect SAT solvers often present an exponential behavior as Figure 5 shows.

For these reasons the standard previously described technique proved to be quite inefficient. On the contrary we do have an estimate of the maximum latency, which is equal to the number of operations in the CDFG. This suggests a second strategy, namely starting from the highest bound and decreasing it in order to find the first unsatisfiable instance. The drawback of this method is that the estimate of the maximum latency can be extremely inaccurate. As a direct consequence, we propose a solution adopting a binary search. Starting with an estimate of the optimal latency, we create the corresponding CNF problem and call the SAT solver giving it a (small) time limit. Accordingly to the result produced by the solver, the estimate of the latency is corrected, and a new bound is tried. Notice that if the SAT solver is unable to solve the CNF problem within the time limit, we consider the instance as unsatisfiable. In general, this might lead to incorrect (i.e., sub-optimal) results, in the sense that a satisfiable instance may be considered as unsatisfiable, but the problem can be solved simply increasing the "unsat" threshold, with an at most linear

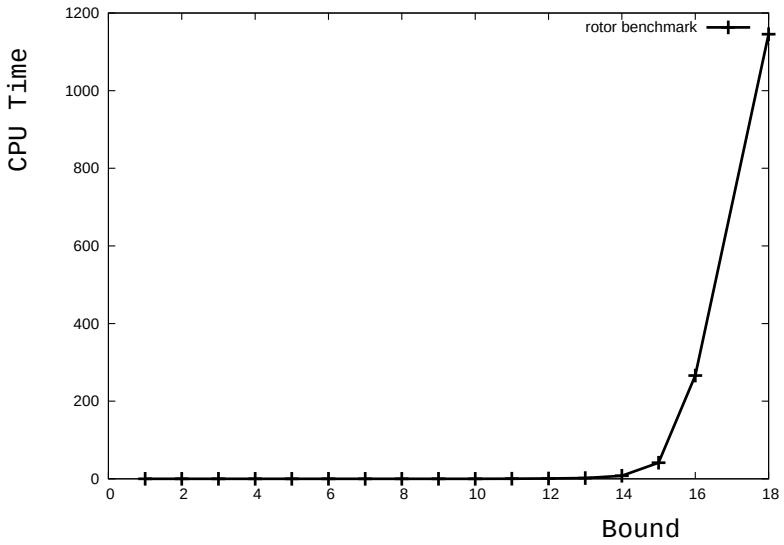

Fig. 5. SAT CPU Time Versus SAT Bound. The data are obtained with a run of the scheduler on the rotor benchmark with two iterates and a resource availability corresponding to the first row of table 2.

loss in performances.

6 Experimental Results

We show experimental results on well known benchmarks [2]. Table 1 shows the complexity of the benchmark set in terms of number of operations, and number of conditions checked.

Circuit	# Operations	# Conditions
rotor	28	3
s2r	48	6
fdct	42	0

Table 1
Circuit Complexity in terms of Number of Operations and Conditions Checked. The data are referred to the acyclic version of the model, i.e., with just one iterate.

We ran our experiments on a 1700 MHz Pentium IV with 1 GByte of main memory. For all the experiments we used BerkMin [12] as SAT engine.

Table 2 summarizes our results. We compare the results obtained with the strategy presented in this paper with the software presented in [2] and locally re-run. More in detail, our data are obtained adopting the binary search (as described in the previous section) with a threshold of two minutes. Notice that all the satisfiable instances were well recognized by the SAT solver; indeed the numbers of scheduled cycles represent the true optimal latencies.

The meaning of columns is the following: # Iterates indicates the number of parallel instances considered (when 1, we refer to the acyclic problem,

Circuit	# Iterates	# Resources	# Cycles	BDD [2]		SAT – This Paper		
				# BDD [nodes]	Time [s]	# Vars	# Clauses	Time [s]
rotor	1	1T,1C,1A	12	74606	0.7	994	4116	0.3
	1	1T,1C,2A	8	74606	0.7	892	4232	0.3
	1	1T,1C,2A,1∗	10	84826	0.8	1142	4174	0.4
	1	1T,1C,2A,2∗	8	84826	0.7	920	4152	0.3
	2	1T,1C,2A,1∗	10	871766	4.6	4749	36047	10.0
	2	1T,1C,2A,2∗	9	1447152	6.8	4392	32873	10.8
	2	1T,1C,2A,3∗	9	1864128	8.2	4428	33026	8.6
	2	1T,1C,3A,2∗	8	415954	2.7	4299	31503	6.7
	3	1T,1C,2A,1∗	12	18635148	1524.0	9573	72436	118.9
	3	1T,1C,2A,2∗	12	OVF	–	9861	73612	308.9
	3	1T,1C,2A,3∗	12	OVF	–	9957	74008	288.3
	3	1T,1C,3A,2∗	9	OVF	–	8229	59803	37.6
s2r	1	1T,1C,2A,1∗	10	1006670	5.9	2532	12484	1.8
	1	1T,1C,3A,2∗	9	532462	4.1	2788	14774	2.2
	1	1T,–C,2A,2∗	8	411866	3.0	4749	12832	1.7
	2	1T,1C,2A,1∗	13	OVF	–	12158	109623	328.6
	2	1T,1C,3A,2∗	10	OVF	–	10839	87817	66.8
	2	1T,–C,2A,2∗	10	OVF	–	10539	87137	62.9
fdct	1	1+,1−,1∗	19	306600	1.5	2775	8362	133.7
	1	1+,1−,2∗	13	200312	1.2	2138	6772	1.1
	2	1+,1−,1∗	32	–	OVF	19121	223433	522.0
	2	1+,1−,2∗	26	–	OVF	17103	188719	454.6

Table 2
Schedule Results. Terminology for columns # Resources: ADD=+, ALU=A, COMPARATOR=C, SUB=−, MULT=∗, LookUpTable=T. MULT is a two-time steps pipelined multiplier (when not present, multiplications are performed by the ALU). All other resources are single time step. OVF indicates overflow (in terms of memory or CPU time). We use a time limit equal to 1 hour and a memory limit equal to 500 MBytes.

otherwise we are handling a looping behavior); column # Resources indicates the number and type of resources allowed; # Cycles is the final solution in term of scheduled cycles. For each experiment we report the data obtained with [2], i.e., the number of BDD nodes and the CPU time required, and with our method (number of variables and clauses generated for the CNF problem corresponding to the solution, and the total CPU time).

Overall, we can make the following observations. For acyclic problems, the times required by the two compared methods are quite similar (with only one exception, the first experiment for fdct). However, when we move to looping behaviors, while the method used in [2] becomes unfeasible, our strategy still

produces the optimal result in a limited amount of time. These experiments demonstrate that our solution can be very effective.

7 Conclusions and Future Work

We present a new approach for symbolic scheduling based on a new problem formulation and the use of SAT solvers and BMC verification methodology.

Experimental results on DFGs and CDFGs show that our solution can be very effective and competitive with symbolic BDD-based techniques.

Future work will include investigation of better strategies for the CNF problem generation and solution searching.

References

[1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, "Hardware-Software Co-design of Embedded Systems – The POLIS Approach," Kluwer Academic Publishers, 1997.

[2] S. Haynal, "Automata-Based Symbolic Scheduling," PhD thesis, University of California Santa Barbara, Dec. 2000.

[3] S. Haynal and F. Brewer, "Efficient Encoding for Exact Symbolic Automata–Based Scheduling," Proc. IEEE ICCAD'98, pages 477–481, San Jose, California, Nov. 1998.

[4] S. Haynal and F. Brewer, "Automata-Based Scheduling for Looping DFGs", Internal Report EC99_14, Oct. 1999.

[5] http://ftp.ics.uci.edu/pub/hlsynth/{HLSynth92,HLSynth95}.

[6] http://www.synopsys.com/products/logic/design_compiler.html.

[7] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis," IEEE Trans. on Computer-Aided Design, 10:464–475, Apr. 1991.

[8] K. Khouri, G. Lakkshminarayana, and N. Jha, "High-level synthesis of low-power control-flow intensive circuits," IEEE Trans. on Computer-Aided Design, 18(12):1715–1729, Dec. 1999.

[9] A. C. Parker, J. T. Pizzarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," Proc. IEEE/ACM ICCAD'91, pages 461–466, Las Vegas, June 1986.

[10] P. G. Paulin and J. P. Knight, "Force–Directed Scheduling for the Behavioral Synthesis of ASICs," IEEE Trans. on Computer-Aided Design, 8:661–679, June 1989.

[11] I. Radivojevic and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling," IEEE Trans. on Computer-Aided Design, C–15(1):45–57, Jan. 1996.

[12] E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust SAT-Solver," Proc. IEEE/ACM DATE'02, pages 142–149 Paris, Feb. 2002.

[13] G. Cabodi and S. Nocco and S. Quer, "Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals," Proc. IEEE/ACM DATE 2003, pages 898–903, Munich, Germany, March 2003.