

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. 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 $\mathbf{T}(\mathbf{s}, \mathbf{s}')$, the set of initial states by a formula $\mathbf{I}(\mathbf{s})$, and further denote the safety property by $\mathbf{P}(\mathbf{s})$. We will use \mathbf{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 $\mathbf{I}(\mathbf{s}_n)$, $\mathbf{P}(\mathbf{s}_n)$, and $\mathbf{T}(\mathbf{s}_n, \mathbf{s}_{n+1})$.

2.1 The SAT problem

Let *Bool* denote the *boolean* domain $\{0, 1\}$, and $\text{Vars} := \{x_0, x_1, x_2, \dots\}$ be a finite set of boolean variables. A *literal* is a boolean variable x_i or a negated boolean variable \bar{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 $\text{Vars} \rightarrow \text{Bool}$. A literal x_i is said to be satisfied by a valuation if its variable is mapped to 1; a literal \bar{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.

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 $\{ \{\bar{p}, x\}, \{\bar{p}, y\}, \{p, \bar{x}, \bar{y}\} \}$.

2.2 Temporal Induction

This section briefly summarizes the verification technique *temporal induction* presented in [SSS00].¹ The word “temporal” suggests that the induction is car-

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.

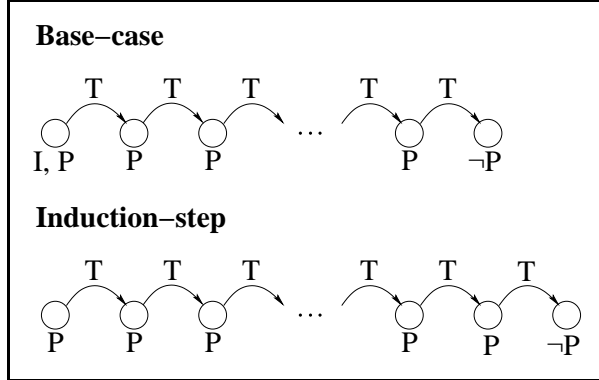


Fig. 1. If the n -th *base-case* is unsatisfiable, it should be read as “There exists no n -step path to a state violating the property, assuming the property holds the first $n - 1$ steps.” If the n -th *induction-step* is unsatisfiable, it should be read as “Following an n -step trace where the property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

ried 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:

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

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 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  $n \in 0..∞$  do
  if (satisfiable([Basen]))
    return PROPERTY FAILS
  if ( $\neg$ satisfiable([Stepn]  $\cup$  [Uniquen]))
    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 $\{\bar{x}, \bar{y}\}$ 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

² This includes SAT-solvers such as: GRASP, SATO, ZCHAFF, LIMMAT, BERKMIN, and the authors’ own solvers SATNIK and SATZOO.

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 *learned 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.

```

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 2.2 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.

Algorithm 2 “Extending base”. **Algorithm 3 “Extending step”.**

<pre> addClauses([I₀]) for n ∈ 0..∞ do addClauses([P_n]^{p_n}) solve({$\overline{p_n}$}) if (SATISFIABLE) return PROPERTY FAILS addClause({p_n}) addClauses([T_n]) </pre>	<pre> addClauses($\overline{[P_0]}$) 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]) </pre>
---	--

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 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 2.2, 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.

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 4 “Zig-zag”.

```

addClauses([I0]z)           – z is the definition literal for I0
for n ∈ 0..∞ do
  addClauses([Pn]pn)       – pn is the definition literal for Pn
  solve({ $\overline{p_n}$ })         – step: do not include I0
  if (UNSATISFIABLE)         – Pn must hold!
    return PROPERTY HOLDS
  solve({z,  $\overline{p_n}$ })       – base-case: include I0
  if (SATISFIABLE)           – counter-example found!
    return PROPERTY FAILS
  addClause({pn})           – assert Pn from now on
  addClauses([Tn])           – assert transition from sn to sn+1
  for i ∈ 0..n-1 do
    addClauses([si ≠ sn]) – add uniqueness constraints

```

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 **B** (bad) for $\overline{\mathbf{P}}$ we can view the set of possible shortest counter-examples pictorially:

```

length 0:           IB
length 1:           I $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$ 
length 2:           I $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$ 
length 3:           I $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$ 
                    ...
length n:           I $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$  ...  $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$   $\overline{\mathbf{T}}$   $\overline{\mathbf{B}}$ 

```

Each line depicting a (shortest) counter-example corresponds to a conjunction of constraints ($\mathbf{I}_0 \wedge \mathbf{T}_0 \wedge \overline{\mathbf{B}}_1 \wedge \overline{\mathbf{I}}_1 \wedge \mathbf{T}_1 \wedge \dots$). 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.:

$$(1) \quad \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}} \overset{\mathbf{T}}{\frown} \dots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{B}}$$

or

$$(2) \quad \overline{\mathbf{I}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \dots \overset{\mathbf{T}}{\frown} \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}$$

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.

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)

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

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.

4.2 Improving the Unique States Requirement

The uniqueness constraints described in section 2.2 and used in *Algorithm 1, 3* and *4* require each pair of states to be different. These requirements are *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 **I** and **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*.

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

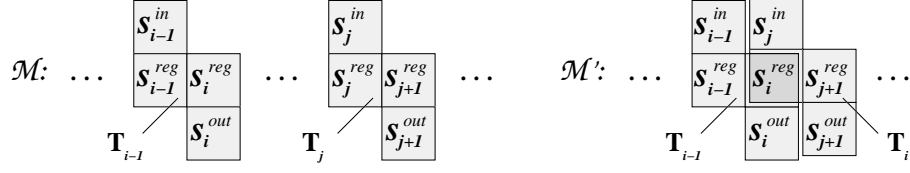


Fig. 2. The picture shows the contraction of the counter-example \mathcal{M} to \mathcal{M}' . The state variables constrained by the transition relations at the point of “gluing” are printed in the boxes; the remaining trace is represented by the “...”.

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:

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

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

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

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

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

\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 s_i are \mathbf{T}_{i-1} and \mathbf{T}_i . But \mathbf{T}_{i-1} does not contain any variables from s_i^{in} , so letting $\mathcal{M}(s_i^{in}) \neq \mathcal{M}'(s_i^{in})$ cannot make \mathbf{T}_{i-1} false in \mathcal{M}' . Similarly for \mathbf{T}_i which does not contain any variables from s_i^{out} . Finally $\mathcal{M}(s_i^{reg}) = \mathcal{M}(s_j^{reg})$, so indeed \mathcal{M}' must be a model for the constraints \mathbf{T}_{i-1} and \mathbf{T}_i . \square

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 TIP⁶ 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* available 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.

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 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.

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/>

TIP was able to solve 6 instances where BDD-based verification failed, showing that induction may be a valuable complementary method.⁷

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.

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.

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.

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.

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
<i>cmu:periodic</i>	97	70.7	> 600	10.7	141.8	80.9	> 600	> 600
<i>eijk:S208c</i>	259	448.0	> 600	> 600	> 600	> 600	> 600	> 600
<i>eijk:S208o</i>	258	483.2	> 600	> 600	> 600	> 600	564.2	> 600
<i>eijk:S208</i>	259	436.7	> 600	> 600	> 600	> 600	503.7	> 600
<i>eijk:S298</i>	59	27.7	> 600	34.9	96.2	62.9	316.1	> 600
<i>eijk:S510</i>	11	5.2	8.0	0.5	0.9	5.9	7.4	10.1
<i>eijk:S820</i>	12	6.1	22.9	6.4	12.5	12.6	20.2	30.1
<i>eijk:S832</i>	12	7.6	28.2	5.8	12.9	13.4	25.1	35.2
<i>eijk:S953</i>	8	1.7	4.2	0.1	0.2	1.9	4.2	4.4
<i>ken:oop₁</i>	30	39.4	> 600	0.3	7.4	39.9	492.0	254.0
<i>nusmv:guidance₁</i>	11	2.8	10.2	0.8	3.4	3.5	3.9	11.1
<i>nusmv:guidance₇</i>	28	120.3	> 600	315.0	> 600	438.9	> 600	> 600
<i>nusmv:tcas₂</i>	7	1.3	3.1	0.2	0.3	1.5	1.9	4.3
<i>nusmv:tcas₃</i>	6	1.3	3.3	0.0	0.1	1.3	1.8	3.2
<i>texas:parse_{sys}₂</i>	4	12.2	13.5	0.2	0.2	14.7	12.5	7.8
<i>vis:prodcell₁₂</i>	30	256.6	> 600	112.8	445.5	367.3	> 600	> 600
<i>vis:prodcell₁₃</i>	9	4.6	12.4	0.1	0.6	4.8	3.7	14.7
<i>vis:prodcell₁₄</i>	17	31.3	185.1	7.3	14.2	38.7	52.3	219.9
<i>vis:prodcell₁₅</i>	24	109.3	> 600	23.0	80.1	132.4	216.7	> 600
<i>vis:prodcell₁₆</i>	6	2.1	4.1	0.0	0.1	2.1	1.2	4.7
<i>vis:prodcell₁₇</i>	28	211.3	> 600	52.4	277.5	265.0	> 600	> 600
<i>vis:prodcell₁₈</i>	14	21.4	117.9	0.4	3.2	21.8	28.6	128.9
<i>vis:prodcell₁₉</i>	23	61.6	457.0	23.4	86.0	85.0	178.5	> 600
<i>vis:prodcell₂₄</i>	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
<i>nusmv</i> :tcas ₁	11	3.6	3.7	5.0
<i>nusmv</i> :tcas ₄	15	9.7	9.7	18.2
<i>nusmv</i> :tcas ₅	24	48.7	40.1	125.2
<i>nusmv</i> :tcas ₆	17	13.6	13.5	38.2
<i>texas</i> :parsesys ₁	10	9.3	0.8	1.1
<i>texas</i> :parsesys ₃	9	3.3	0.7	0.9
<i>texas</i> :two-proc ₂	16	4.7	1.0	2.9
<i>texas</i> :two-proc ₄	20	20.9	1.8	9.1
<i>vis</i> :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
<i>cmu</i> :periodic	97	70.7	120.4	0	4656	455k	908k	15k	14k
<i>eijk</i> :S208	259	436.7	[> 600]	258	[>20000]	186k	-	76k	-
<i>eijk</i> :S298	59	27.7	66.6	114	1653	69k	296k	24k	25k
<i>ken</i> :oop ₁	30	39.4	50.4	113	406	67k	101k	32k	30k
<i>nusmv</i> :guidance ₇	28	120.3	66.9	0	378	151k	276k	56k	28k
<i>vis</i> :prodcell ₁₂	30	256.6	252.7	0	406	346k	439k	48k	43k
<i>vis</i> :prodcell ₁₄	17	31.3	41.7	0	120	189k	217k	11k	13k
<i>vis</i> :prodcell ₁₅	24	109.3	134.3	0	253	273k	330k	29k	29k
<i>vis</i> :prodcell ₁₇	28	211.3	253.6	0	351	322k	400k	45k	46k
<i>vis</i> :prodcell ₁₈	14	21.4	25.5	0	78	153k	171k	10k	10k
<i>vis</i> :prodcell ₁₉	23	61.6	71.9	0	231	260k	311k	18k	18k
<i>vis</i> :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.

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].

Related techniques for proving upper bounds for BMC are presented in [KS03] (computing the recurrence diameter) and [BKA02] (approximating the diameter by structural analysis). In particular, the authors of [KS03] suggest another solution to the quadratic blow-up of uniqueness constraints by adding a sorting network for the state variables to the SAT-problem.

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 underlying 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

- [BKA02] J. Baumgartner, A. Kuehlmann, J. Abraham “**Property Checking via Structural Analysis**” in *CAV 2002*, LNCS:2404, Springer-Verlag.
- [BC00] P. Bjesse, K. Claessen. “**SAT-based Verification without State Space Traversal**” in *FMCAD 2000*, LNCS:1954, Springer-Verlag.
- [BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. “**Symbolic model checking without BDDs**” in *TACAS 1999*, LNCS:1579, Springer.
- [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 *CAV 1999*, LNCS:1633.
- [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. Copt, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y. Vardi. “**Benefits of bounded model checking at an industrial setting**” in *CAV 2001*, LNCS:2102.
- [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.
- [KS03] D. Kroening, O. Strichman “**Efficient Computation of Recurrence Diameters**” in *VMCAI 2003* LNCS:2575, Springer-Verlag 2003.
- [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 *DAC 2001*.
- [Stri00] O. Strichman “**Tuning SAT checkers for Bounded Model Checking**” in *CAV 2000*, LNCS:1855, Springer-Verlag.
- [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 *FMCAD 2000*, LNCS:1954.
- [WKS01] J. Whittemore, J. Kim, K. Sakallah. “**SATIRE: A New Incremental Satisfiability Engine**” in *DAC 2001*, ACM Press.