Simultaneous SAT-Based Model Checking of Safety Properties

Zurab Khasidashvili[1], Alexander Nadel[1,2], Amit Palti[1], and Ziyad Hanna[1]

[1] Design Technology Solutions, INTEL Corporation
{zurab.khasidashvili, alexander.nadel, amit.palti,
ziyad.hanna}@intel.com
[2] Department of Computer Science,
Tel Aviv University, Ramat Aviv, Israel

Abstract. We present several algorithms for simultaneous SAT (propositional satisfiability) based model checking of safety properties. More precisely, we focus on Bounded Model Checking and Temporal Induction methods for simultaneously verifying multiple safety properties on the same model. The most efficient among our proposed algorithms for model checking are based on a simultaneous propositional satisfiability procedure (SSAT for short), which we design for solving related propositional objectives simultaneously, by sharing the learned clauses and the search. The SSAT algorithm is fully incremental in the sense that all clauses learned while solving one objective can be reused for the remaining objectives. Furthermore, our SSAT algorithm ensures that the SSAT solver will never re-visit the same sub-space during the search, even if there are several satisfiability objectives, hence one traversal of the search space is enough. Finally, in SSAT all SAT objectives are watched simultaneously, thus we can solve several other SAT objectives when the search is oriented to solve a particular SAT objective first. Experimental results on Intel designs demonstrate that our new algorithms can be orders of magnitude faster than the previously known techniques in this domain.

1 Introduction

Bounded Model Checking (BMC) [BCCZ99, BCC+03] is a SAT (or satisfiability) [DLL62] based verification technique, well suited for finding counter-examples to a given safety property P, in a transition system. It arose as a complementary approach to BDD-based [Bry86] *Symbolic Model Checking* technique [McM93], and is increasingly adopted by the industry [PBG05]. The idea of BMC is to unroll the transition system to k time steps, and search using a SAT solver for a state transition path of length less or equal to k, starting with an initial state and ending in a state violating the property. We recall that a SAT solver searches for a satisfying assignment to a Boolean formula written in CNF form; such a formula is represented as a set of clauses, a clause being a disjunction of literals, where a literal is a Boolean variable or its negation.

S. Ur, E. Bin, and Y. Wolfsthal (Eds.): Haifa Verification Conf. 2005, pp. 56 – 75, 2006.

We restrict ourselves to safety properties written as ***AG****p* in Computation Tree Logic (CTL, [CGP99]). Such properties are often called *invariants*. Proving a property *P* using BMC technique means showing that there is no counter-example to *P* of the length less or equal to the diameter of the system, i.e., the maximum length of a shortest (thus loop-free) path between any two states. From practical point of view, BMC is an incomplete technique in that it can rarely prove a property arising from an industrial application of software or hardware verification, since the diameter for such systems is too large to handle by current SAT solvers. A practical, complete SAT-based model checking method was proposed by Sheeran.et. al. [SSS00] as a (temporal) induction method, allowing proving safety properties by means of unrolling to much lower depths than the diameter. Roughly, BMC in this method corresponds to the base of temporal induction, and the induction step, at depth *m*, attempts to prove that there is no state transition path $s_0,\ldots,s_m, s_{m+1}$ such that *P* holds in all but the last state (here s_0 needs not be an initial state). Once such an *m* is found, and it has been shown in the base of induction that there is no counter-example to the property of length *m* or less, the property *P* is proven valid at all states reachable from the initial state.

Several usability enhancements have been proposed in the literature to the above methods, boosting the capacity to handle larger systems and complex properties, and faster. These enhancements are very important for successful application of the methods in practice. Here we review briefly two enhancements that are most relevant to our work. For more information, we refer the reader to a recent survey of SAT-based model checking [PBG05].

In a BMC run, to avoid unnecessary unrolling of the transition relation, one starts with low bounds *k*, and if no counter example is found for the property of length smaller or equal to *k*, the bound *k* is increased, repeatedly, till it reaches the diameter or a maximal user given value for the bound. Therefore a BMC run involves a number of calls to the SAT solver. Similarly, proving the induction step in temporal induction method needs several calls to the SAT solver, with increasing bounds *m*. These SAT instances are closely related, and the idea of *incremental SAT solving* in BMC and induction (as well as in other SAT applications), proposed independently by Strichman [Str04] and Whittemore et al [WKS01], is in re-using *pervasive* learned conflict clauses across consecutive calls to the SAT solver. Here pervasive learned clauses are logical consequences of all involved SAT instances, thus adding them to the clause set permanently is safe. Eén and Sörensson [ES03] extended this approach to temporal induction, and proposed a simple interface to a SAT solver enabling incremental BMC and induction schemes where all conflict clauses are pervasive and can be re-used.

In typical industrial model checking applications, one needs to prove a number of properties on the same model. Since several properties may share the "cone of influence" in the model, (dis)proving several properties in one model-checking session may yield a significant speedup. To the best of our knowledge, Fraer et al. [FIK+02] were the first to propose an extension to the classic BMC and the induction method allowing to *simultaneously* check a number of safety properties $P_1,\ldots,P_n$ on the same model.

Here we propose a number of new algorithms for simultaneous SAT-based model checking of multiple safety properties, which strengthen the method of [FIK+02].

Incrementality through verification depths is one source of incremental SAT-based model checking [Str04, WKS01, ES03]. Our algorithms are *double-incremental*, meaning that learned clauses of the SAT solver can be reused across depths, as well as across the properties at every depth. The most efficient among our algorithms use a *simultaneous SAT solver* (SSAT), which is able to resolve several objectives related to the same instance *in one traversal* of the search space. In SSAT, besides a selected, *currently watched* objective, one actually watches all unresolved objectives as well, and can falsify or prove them valid during the search oriented to solve the currently watched objective. Because of these "one traversal" and "all watched" principles, our algorithm is more efficient than previous approaches to fully incremental SAT solving which, like SSAT, allowed reusing *all* learned conflict clauses [GN01, ES03]. We will discuss these approaches in detail in a related work section and will provide experimental results to demonstrate the superiority of the SSAT approach.

The paper is organized as follows: in the next section, we will give a short introduction into modern DPLL-based algorithms. In Section 3, we describe SSAT algorithm and its implementation on top of a DPLL-based propositional SAT solver. In Section 4, we compare the SSAT algorithm with previous approaches to incremental solving of related satisfiability objectives. As one can see, sections 2-4 are dedicated to propositional satisfiability checking. In Section 5, we propose several new, double-incremental methods for simultaneous model checking of safety properties based on SAT algorithms described in sections 2-4. In Section 6, we present experimental results demonstrating the usefulness of the SSAT approach on series of benchmarks originating from formal property verification and formal equivalence verification of Intel designs. Conclusions appear in Section 7.

2 The Basic DPLL Algorithm in Modern SAT Solvers

The DPLL algorithm [DP60, DLL62] is the basic backtrack search algorithm for SAT. We briefly describe the functionality of modern DPLL-based SAT solvers, referring the interested reader to [LM02] or [Nad02] for a more detailed description.

Most of the modern SAT solvers enhance the DPLL algorithm by the so-called Boolean constraint propagation (BCP) [ZM88], conflict driven learning [MS99], [ZMM+01] and search restarts [GSK98]. The SAT solver receives as input a formula in Conjunctive Normal Form (CNF), represented as a set of clauses, each clause being a disjunction of literals, where a literal is a Boolean variable or its negation. The solver builds a binary search tree until it either finds an assignment satisfying all the clauses—a *model*, in which case the formula is *satisfiable*; or it explores the whole search space and finds no model, in which case the formula is *unsatisfiable*. Note that some of the variables in a model may be *don't cares*, meaning that any assignment to these variables still yields a model of the CNF formula.

At each node of the search tree the solver performs one of the following steps:

1. It chooses and assigns the next decision literal and propagates its value using BCP. A unit clause is a clause having all but one literal assigned *false*, while the remaining literal *l* is unassigned. Observe that *l* must be assigned *true* in order to satisfy the formula; this operation is often referred to as the *unit clause rule* and

l = *true* is referred to as an *implied* assignment. BCP identifies unit clauses and repeatedly applies the unit clause rule until either:

- No more unit clauses exist. In this case, the solver checks whether all the clauses are satisfied. If they are, we have found a model and the formula is satisfiable, otherwise the solver is looking for the next decision literal;
- A variable exists that must be assigned both *false* and *true* in two different unit clauses, in which case we say that a *conflict* is discovered.

2. If a conflict is discovered, the solver adds one or more *conflict clauses* to the formula. A conflict clause is a new clause that prevents the set of assignments that lead to the conflict from reappearing again during the subsequent search [MS99]. Then, if a literal y exists such that it is sufficient to flip its value in order to resolve the conflict, the solver backtracks and flips the value of y; otherwise the formula is unsatisfiable. The former case is referred to as a *local conflict* and the latter case is referred to as a *global conflict*. For our purposes, it is also important to mention that during conflict analysis one or more literals may be discovered to be *globally true*, that is, they must be assigned *true* independently of other variable values. This happens every time when a new conflict clause containing exactly one literal l is learned. The literal l as well as all the literals assigned as a result of BCP following the assignment l = *true* are globally true.
3. Once in a while the solver restarts the search, keeping all or some of the learned conflict clauses [GSK98].

We demonstrate the above concepts on a simple example taken from [Str04].

Example 1: Consider the following set of clauses $\{c_1,c_2,c_3,c_4\}$, where

$c_1 = \neg x_1 \vee x_2$
$c_2 = \neg x_1 \vee x_3 \vee x_5$
$c_3 = \neg x_2 \vee x_4$
$c_4 = \neg x_3 \vee \neg x_4$

Assume the current assignment is x_5 = *false* (*i*.e., x_5 is assigned *false*), and a new decision assignment is x_1 = *true*. The resulted implication graph is shown in Figure 1: applying BCP leads to conflicting assignments x_4 = *true* and x_4 = *false*. The clauses $\neg x_1 \vee \neg x_3$; $\neg x_1 \vee x_5$ are examples of conflict clauses, and a subset of conflict clauses is kept as *learned* conflict clauses.

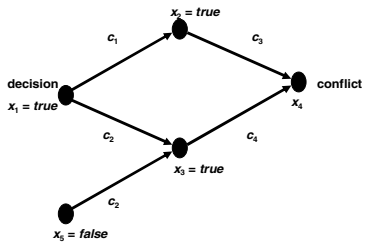

Fig. 1. An implication graph

3 SSAT Implementation Within a DPLL-Based SAT Solver

Now we describe the simultaneous propositional satisfiability algorithm, denoted as *SSAT*. In addition to the input formula (the SAT instance, or CNF instance), SSAT receives as a parameter a list of proof objective literals (PO literals, or POs for short). The POs must be proven *falsifiable* or *valid*. We require the variables of the PO literals to occur (positively or negatively, or both) in the CNF instance.[1]

Example 1 (continued): Assume our SAT instance consists of the same four clauses $\{c_1, c_2, c_3, c_4\}$, and assume our PO literals (or simply POs) are $PO_1 = \neg x_1$, $PO_2 = x_5$, and $PO_3 = x_2$. One can verify that all of the POs are falsifiable – there is a model for the SAT instance (i.e., an assignment satisfying the instance) where $\neg x_1$ is assigned *false* (thus x_1 is assigned *true*), there is a model where x_5 is assigned *false*, and a model where x_2 is assigned *false*. For example, the (partial) assignment $x_1 = false$, $x_3 = false$, $x_4 = true$, $x_5 = false$ is a model for $\{c_1, c_2, c_3, c_4\}$ in which PO_2 is falsified. Note that x_2 is unassigned in the model – x_2 is a *don't care variable* since assigning any of the truth values to x_2 yields a model for $\{c_1, c_2, c_3, c_4\}$. Assigning $x_2 = false$ yields a model in which PO_3 is falsified. The partial assignment $x_1 = false$, $x_3 = false$, $x_4 = true$ is another model (with more don't cares) for $\{c_1, c_2, c_3, c_4\}$.

A straightforward way to implement SSAT is as follows: (1) proceed with a regular DPLL-based search; (2) when a model is discovered, mark all the POs that are assigned *false* in the model as *falsifiable*; (3) as soon as all the search space is explored, mark all the unmarked POs *valid* and exit. However, there is a major problem with this solution: the number of models might be very large and therefore it is extremely inefficient to visit each of them during the search. Moreover, if a SAT solver uses search restarts (as do most of the state-of-the-art solvers), the algorithm might never finish, since the same models could be rediscovered after each restart. One solution could be adding clauses preventing the rediscovery of each model, but this might lead to memory explosion. We propose the following solution to this problem.

We always maintain a PO literal that we are trying to falsify, called the *currently watched PO* (*CWPO*). At the beginning of the search we set *CWPO* to be any PO literal. At any stage of the search, prior to invoking a generic decision heuristic, we assign *CWPO* the value *false*, if not already assigned. The *CWPO* ceases to be the currently watched PO under two circumstances: (1) When a model containing *CWPO* = *false* is discovered, in which case we mark as *falsifiable* the *CWPO* as well as all the POs that are assigned *false* (or are don't care literals) in the model; (2) When *CWPO* is discovered to be globally *true*, in which case we mark the *CWPO* as well all other globally valid POs (if any) as *valid*. On both occasions, we check whether there exists a PO *l* that has not been discovered *valid* or *falsifiable*, in which case we set *CWPO* to *l*, otherwise the algorithm halts. This simple adjustment ensures that: (a) the number of discovered models is at most the number of POs; (b) a model is never rediscovered even if search restarts are used. Indeed, after encountering a model we

[1] We expect that POs are related to the instance; the definition of the POs in terms of variables in the instance can be included as a part of the CNF instance; thus the above requirement is not a restriction from application point of view.

always choose a *CWPO* that has not been *false* under any model and assign it the value *false*. This guarantees that any model will be different from all the previously discovered models. In addition, since the number of *CWPO*s is at most equal to the number of POs, the number of discovered models is at most equal to the number of POs. Also, our algorithm ensures that every PO is visited during new *CWPO* selection and thus every PO is marked *valid* or *falsifiable* after SSAT terminates.

The SSAT algorithm is presented in Figure 2. First, SSAT chooses a *CWPO*. Then, it enters a loop that terminates only when all the POs are proven to be either *falsifiable* or *valid*. Within the loop, SSAT first checks whether the current *CWPO* has already received a value. If it has, then a new *CWPO* is selected and assigned *false*. If all POs are resolved, the algorithm terminates. If the current *CWPO* has not been resolved yet, a new decision literal is picked using a generic decision heuristic. At the next stage, a conflict analysis loop is entered. After BCP, SSAT marks any PO that was found to be globally *true* as *valid*. Then, *SSAT* checks what the status of the formula is after BCP. If a global conflict has been discovered, that is, all the assignment space has been explored, the algorithm marks all the unmarked POs as

```
SSAT ([PO1,..,POn], cnf_instance) {
   Literal CWPO = any PO literal;
   while (1) {
      if (CWPO is valid or falsifiable) {
         if (all the POs are valid or falsifiable) Return;
         CWPO = any PO literal that is neither valid nor falsifiable;
         Assign CWPO = false;
      } else {
         Assign choose_decision_literal();
      }
      do {
         status = BCP();
         Mark any PO literal that is discovered to be globally true as valid;
         if (status == global_conflict) {
            Mark all unmarked PO literals valid; Return;
         }
         if (status == model) {
            Mark any falsified and don't care PO literal falsifiable;
            Unassign all the literals that are not globally true;
         }
         if (status == local_conflict) {
           Add a conflict clause; Backtrack;
           Assign literal that must be flipped following conflict analysis;
         }
      } while (status == local_conflict);
   }
}
```

Fig. 2. SSAT pseudo-algorithm

valid (since after exploring the whole search tree, we discovered that no model falsifies them) and halts. If a model has been discovered, SSAT marks as *falsifiable* all the POs that are assigned *false* or are don't cares in the model and unassigns all the literals except ones that are globally *true*. Observe that in this case the algorithm exits the conflict analysis loop and picks the next *CWPO* during a new iteration of the global loop. Finally, if a local conflict has been encountered, SSAT backtracks and flips the value of a certain literal. Observe that in this case the algorithm goes

on with the conflict analysis loop. Notice that it is safe to use restarts in the SSAT algorithm.

4 Comparing Simultaneous SAT Algorithm with Previous Work

The incremental satisfiability technique proposed in [MS97, WKS01, Str04] is based on identifying and reusing the *pervasive conflict clauses* encountered by the SAT solver during the search for a satisfying assignment to a given CNF formula. When one is trying to solve related SAT problems, the clauses occurring in the CNF formulas that are to be checked for satisfiability, which we will call the satisfiability objectives, can be divided into two classes: the clauses that are common to all satisfiability objectives will be called *pervasive clauses*, and the remaining clauses will be called *temporal clauses*. Then the conflict clauses that can be derived solely from the pervasive clauses are *pervasive conflict clauses*, and can be used for resolving each satisfiability objective. Experimental results in [WKS01, Str04] amply demonstrate that pervasive conflict clauses can significantly accelerate solving families of related SAT objectives. We will refer to this approach as PISAT approach.

To understand the differences between our SSAT approach and the PISAT approach, here we explain on an example the definition of pervasive conflict clauses and their usage in incremental SAT solving, as proposed in [Str04].

Example 1 (continued): Suppose again we have the same SAT formula consisting of clauses $\{c_1, c_2, c_3, c_4\}$. Further, define clauses $c_5 = x_1$, $c_6 = \neg x_5$, and $c_7 = \neg x_2$, and assume we are interested in solving the following three SAT instances:

(1) $\{c_1, c_2, c_3, c_4, c_5\}$
(2) $\{c_1, c_2, c_3, c_4, c_6\}$
(3) $\{c_1, c_2, c_3, c_4, c_7\}$

The incremental SAT solving approach proposed in [Str04] is as follows: One observes that clauses $\{c_1, c_2, c_3, c_4\}$ are common to all three SAT problems, and clauses c_5, c_6 and c_7 are unique to particular SAT instances (1), (2) and (3), respectively. When solving the instance (1), one marks clauses c_1, c_2, c_3, and c_4, and for every conflict clause c encountered during the SAT search, if all clauses leading to the conflict are already marked, then one marks c as well. Note that all pervasive conflict clauses are logical consequences of $\{c_1, c_2, c_3, c_4\}$, thus the satisfiability of (2) and (3) will remain unaffected if the pervasive conflict clauses are added to instances (2) and (3).

Suppose when solving instance (1), the SAT solver chooses first the assignment $x_5 = \mathit{false}$.[2] From this assignment, using clause c_5, BCP will force implied assignment $x_1 = \mathit{true}$, and further iterations of unit clause rule in BCP will lead to the discovery of

[2] Most of the modern SAT solvers would start with BCP, and BCP in our example would find a model. We have chosen to start with assignment x5 = *false* for demonstration purposes, and this allows us to reuse example from [Str04] (and to keep the presentation simple).

a conflict clause $\neg x_1 \vee x_5$ (as shown before, see Figure 1). Since clause c_5 is responsible for that conflict, the conflict clause will not be marked as pervasive, and its usage is not allowed during SAT search for instance (2) (and instance (3)). When trying to resolve instance (2), the SAT solver may again choose to assign $x_5 = \mathit{false}$ and then $x_1 = \mathit{true}$, and discover the same conflict clause again – a duplication of work, which is desired to avoid.

The reader may have noticed that solving SAT problems (1), (2) and (3) corresponds to solving the validity of POs $PO_1 = \neg x_1$, $PO_2 = x_5$, and $PO_3 = x_2$, respectively, from our running Example 1. In the *SSAT* algorithm, there is no need to distinguish between pervasive and other conflict clauses – all conflict clauses are re-usable till the end of the *SSAT* search. Thus any conflict clause can be added to the original clause set without affecting any of the POs' status, and no such conflict clause will be encountered twice in a *SSAT* search. The following is a possible scenario of a *SSAT* run on our running example: Suppose *SSAT* selected PO_2 as the first currently watched PO. Then x_5 is assigned *false*. BCP yields no implied assignments, and *SSAT* may chose $x_1 = \mathit{true}$ as the next decision. BCP will then discover the conflict clause $\neg x_1 \vee x_5$. A clever decision here is to flip the conflicting assignment of x_1, and assign $x_1 = \mathit{false}$. This assignment satisfies clauses c_1 and c_2. *SSAT* may then choose $x_4 = \mathit{true}$ as the next decision. This assignment will satisfy the clause c_3, and BCP will force implied assignment $x_3 = \mathit{false}$ to satisfy c_4 as well. Thus we got a model $x_5 = \mathit{false}$, $x_1 = \mathit{false}$, $x_3 = \mathit{false}$, $x_4 = \mathit{true}$. Variable x_2 is a don't care variable for the discovered model, thus PO_3 can also be declared *falsifiable*. Thus SSAT is left with PO_1; it chooses PO_1 as *CWPO* and assigns it *false* – thus $x_1 = \mathit{true}$. BCP will use the previously discovered conflict clause $\neg x_1 \vee x_5$ to force assignment $x_5 = \mathit{true}$ (here we have used a conflict clause that is not pervasive in the sense of [Str04]); BCP will also imply assignments $x_2 = \mathit{true}$, $x_4 = \mathit{true}$ and $x_3 = \mathit{false}$. SSAT has thus discovered a counter model to validity of PO_1 – $x_1 = \mathit{true}$, $x_5 = \mathit{true}$, $x_2 = \mathit{true}$, $x_4 = \mathit{true}$, $x_3 = \mathit{false}$. *SSAT* will report PO_1 as *falsifiable* and exit.

We have mentioned that SSAT can declare a PO *valid* during the search when it discovers that the PO is *globally true*. This happens when a conflict clause is encountered in which the PO is the unique literal. This can also happen during BCP. In the experimental results section we give data on the valid POs proved in such situations – *all* POs proved valid in SSAT are such POs. Note that in the PISAT approach, a PO can be proved valid if the instance where the PO is assigned *false* is *unsatisfiable* – thus it is necessary to cover the entire search space, while in *SSAT* the POs can be proved valid after a partial traversal of the search space.

It is worth reiterating that in SSAT it is possible to falsify several POs based on the same model. We have seen above a toy example where three POs are falsified based on two models. In the next section we will give experimental data on this as well. This simultaneous falsification feature significantly accelerates the SSAT algorithm. When one works with POs, a similar feature can also be implemented for the PISAT approach based on pervasive conflict clauses. Such a simultaneous falsification feature was indeed activated in the benchmark runs reported in the next section.

Re-usage of certain conflict clauses is the essence of the incremental approach of [MS97, WKS01, Str04]. However, PISAT allows one to reuse only pervasive

conflict clauses; hence any conflict whose associated conflict clause is temporal may reappear while solving the next instance. In another context, Goldberg and Novikov [GN01] proposed a method, which we refer to as GN, allowing one to reuse all conflict clauses when solving multiple POs for a given single propositional instance – thus tracking pervasive clauses becomes redundant. Similarly to our approach, GN maintains a currently watched PO (*CWPO*). It assigns *CWPO* the value *false* prior to assigning values to other variables. From then on, GN treats *CWPO* as a normal decision variable and proceeds with a DPLL-style, backtrack search. If a model is found while exploring the subspace, where *CWPO* is assigned *false*, *CWPO* is *falsifiable*, otherwise it is *valid*. After GN completes checking a certain *CWPO*, it augments the initial formula with all or some of the recorded conflict and uses the described above method to determine the status of the remaining POs.

In contrast with PISAT, GN allows one to reuse every conflict clause recorded while checking a certain PO. Indeed, CWPOs are treated as internal assumptions and every recorded conflict clause is guaranteed to be independent of internal assumptions. This feature is common with our SSAT algorithm; still there are important differences between the GN and SSAT algorithms. Most importantly, SSAT is oriented towards simultaneous solving of the POs – it watches all POs and tries to decide other POs while checking the CWPO, whereas GN treats one PO – the CWPO, at a time. To begin with, suppose that a model, falsifying all the POs, is discovered. In this case, SSAT declares all the POs falsifiable and exits, whereas GN falsifies only the *CWPO* and continues to work to falsify other POs. Generally, each time a model falsifying more than one PO is discovered, GN falsifies only the *CWPO*. Another advantage of SSAT is that it can never rediscover the same model, because it always chooses as a *CWPO* only POs that have not been previously falsified by any model. This allows SSAT to prune the search space in a much more efficient manner. In contrast, GN can reach the same models again and again while checking different POs. We will demonstrate empirically that SSAT is more efficient than GN. It is worth mentioning that [GN01] was not written in the context of BMC. Experimental data section of [GN01] contains benchmarks having only a few hundred variables and clauses. Modern BMC benchmarks are larger by 3-5 orders of magnitude, and it is interesting to see how GN performs compared with PISAT and SSAT on such instances.

Conflict clause re-usage was also proposed by Eén and Sörensson [ES03]. The basic idea is the same as in [GN01]. The POs are considered to be internal assumptions, thus every conflict clause is guaranteed to be globally correct. However, [ES03] treats only the case where one should prove a single PO. The conflict clauses are passed between formulas corresponding to different base and step depths. Roughly, the enhanced API proposed by [ES03] corresponds to our SSAT API. However, [ES03] were concerned with solving one objective at a time and their approach lacks the “one traversal” and “all watched” principles of SSAT. We will refer to the fully incremental approaches of [GN01] and [ES03] as FISAT – indeed, their aim (as stated in the respective papers) is to achieve a maximal re-usage of conflicts, rather that *simultaneous solving* of related objectives.

5 Methods for Simultaneous Bounded Model Checking and Induction

Previous sections were concerned with simultaneous solving of propositional objectives. In this section we propose several new methods for simultaneous model checking of multiple safety properties, using the proposed propositional algorithms. Since BMC corresponds to the base of temporal induction, we will mainly discuss induction algorithms. Let us first briefly recall the induction method of [SSS00]. Let $path(s_0,\ldots,s_k)$, $base(P,k)$, $step(P,k)$, and $loopFree(k)$ denote the following formulas:

$$path(s_0,\ldots,s_k) = Tr(s_0,s_1) \wedge Tr(s_1,s_2) \wedge \ldots \wedge Tr(s_{k-1},s_k) \quad (1)$$

$$base(P,k) = I(s_0) \wedge path(s_0,\ldots,s_k) \wedge P(s_0) \wedge \ldots \wedge P(s_{k-1}) \wedge \neg P(s_k) \quad (2)$$

$$step(P,k) = path(s_0,\ldots,s_{k+1}) \wedge P(s_0) \wedge \ldots \wedge P(s_k) \wedge \neg P(s_{k+1}) \quad (3)$$

$$loopFree(k) = path(s_0,\ldots,s_k) \wedge (\bigwedge_{0 \le i < j \le k} (s_i \neq s_j)) \quad (4)$$

where Tr is a transition relation between states s_0, s_1, … of a Finite State Machine M, and $I(s_0)$ denotes that s_0 is an initial state of M. Then a pseudo-code for the basic induction algorithm for an invariant property P looks as follows:

```
BASIC-TEMPORAL-INDUCTION (P, max_depth) {
 k = 0;
   while ( k ≤ max_depth  ) {
        If ( satisfiable (base(P,k) ) {
         Return "P is falsifiable (counter-example length is k)";
        }
        If  ( unsatisfiable (step(P,k) && loopFree(k) ) {
          Return "P is valid";
        }
        k++;
   }
 Return "P has no counter-example of length max_depth or less";
}
```

Fig. 3. Basic temporal induction scheme

Checking P in the BMC style consists of finding a k such that $base(P,k)$ is *satisfiable*. We then can generate a counter-example (CE) of length k, which is an *error trace* for P. The above induction scheme for verifying P consists of finding a k such that either $base(P,k)$ is *satisfiable* (and a CE will be generated) or $base(P,i)$ and $step(P,k)$ are *unsatisfiable* for $0 \le i \le k$, in which case P is *valid* at all reachable states. The *loopFree* condition is needed for the completeness of the algorithm, but the proofs obtained without this condition remain sound. In the algorithms below, for the simplicity of presentation we omit this condition.

Work [ES03] discusses several variations of this basic induction algorithm. There are several ways to combine base checks with step checks, for example. Further, in

the basic algorithm the depth k is incremented by 1, while larger increments are possible by slight modification of the base and step formulas. In the next section where we describe versions of temporal induction algorithms for simultaneous verification of safety properties, for the simplicity of presentation we will only consider a combination of base and step parts in the style of the basic induction algorithm above, and we will only consider increment 1 in base and step depths. Variations similar to those discussed in [ES03] are possible also for this basic simultaneous induction algorithm.

5.1 The Previous Work on Simultaneous Induction

Fraer et al. [FIK+02] proposed a method for *simultaneously* checking a number of safety properties $P_1,\ldots,P_n$ on the same model. Their idea is to form a *conjunction P* from the properties P_i. If P is *false* at depth 0, a CE to P is a CE for a number of properties P_i. These P_i are reported *falsifiable* (at depth 0), and remaining properties will form a conjunction P'. The same process will be applied to P', repeatedly, till the maximal subset of properties whose conjunction is not *falsifiable* at depth 0 is found. To perform BMC, such properties must be checked for depth 1, and so on. The BMC check terminates when all properties are falsified or the depth limit is reached. For the step, the idea is similar: The aim is to find a maximal subset (which actually is *the* maximal subset) V of yet unresolved properties whose conjunction P* can be proven at current depth k (that is, the corresponding step formula *step*(P*,k) must be *unsatisfiable*). The subset V is found after several iterations of SAT-checking of conjunctions of unresolved properties and eliminating properties that cannot be proven at depth k, by inspecting the models returned by the SAT solver.

The next figure describes a basic induction algorithm for multiple safety properties; here and in the remaining algorithms below, U will denote the list of safety properties to be resolved. Furthermore, in these algorithms we normally use callbacks to report the status of the properties (the callbacks are activated during the run, or after the algorithm terminates). The callbacks may or may not be mentioned explicitly. All algorithms return the list of unresolved properties (remaining from the input list).

```
SIMULTANEOUS-INDUCTION(U, max_depth) {
 k = 0;
   while ( k ≤ max_depth && U != ∅) {
        U = simultaneous_base(U,k);
        If ( U != ∅)
          U = simultaneous_step(U,k);
        k++;
   }
 Return U;
}
```

Fig. 4. A basic simultaneous induction scheme for multiple safety properties

In the induction scheme above, for a depth k, the algorithm *simultaneous_ base*(U,k) checks which of the properties in U are *falsifiable* in the instance unrolled to depth k. This can clearly be done in different ways. The *BASE_CONJUNCTION* algorithm below corresponds to the method in [FIK+02] for performing simultaneous base on properties in U:

```
BASE_CONJUNCTION(U,k) {
P = ⋀U; // the conjunction of all formulas in U
  While ( U != ∅) {
   if( satisfiable( base(P,k) ) ) {
      U = base_conj_callback( M );
      P = ⋀U; }
   else {
      break; }
  }
  Return U;
}
```

Fig. 5. The conjunction method for simultaneous base at depth k

Here M is the model returned by the SAT solver, and *base_conj_callback* checks M: all properties P_j in U whose representative variables at depth k are false in M are reported to the user as *falsifiable* at depth k; the list of remaining P_j is saved as U.

Similarly, the *STEP_CONJUNCTION* algorithm below corresponds to the way *simultaneous_step* procedure is performed in [FIK+02]:

```
STEP_CONJUNCTION(U,k) {
 V = U; // properties we may still prove valid at depth k
 U = ∅; // properties we already know cannot be proven at depth k
  While ( V != ∅ ) {
   P = ⋀V;
   if ( satisfiable( step(P,k) ) {
       (V,U) = step_model_callback(M); }
   else {
      Break;  }
  }
  valid_callback(V);
  Return U;
}
```

Fig. 6. The conjunction method for simultaneous step at depth k

Here M is the model returned by the SAT solver, and *step_model_callback* checks which of the variables representing properties in V at depth $k + 1$ are false in M; such properties are moved from V to U, as we know they cannot be proven at depth k; *valid_callback* will report all properties in V valid to the user (the list V may be empty after *STEP_CONJUNCTION* terminates).

5.2 SSAT-Based Induction

In this subsection we propose several new methods for simultaneous temporal induction for multiple safety properties.

The following *BASE_SSAT*(*U*,*k*) procedure is a way to perform the *simultaneous_base*(*U*,*k*) procedure in the *SIMULTANEOUS-INDUCTION* scheme of Figure 4; it uses the SSAT algorithm:

```
BASE_SSAT(U,k) {
   U = SSAT+( U, base_ssat_callback );
   Return U;
}
```

Fig. 7. The SSAT method for simultaneous base at depth *k*

Here $SSAT^+$ starts by running SSAT; the callback *base_ssat_callback* updates the user every time a property P_j from U gets falsified; finally, the list of remaining properties (properties, proved *valid* by SSAT) will be assigned to U.

To describe simultaneous step algorithms for multiple safety properties, let us define:

$$\underline{step\_ssat}(U,k) = [\neg step(P_1,k),..., \neg step(P_n,k)] \tag{5}$$

$$\underline{step\_hybrid}(U,k) = [step_2(U,k,1),\ldots,step_2(U,k,n)] \tag{6}$$

where both $\underline{step\_hybrid}(U,k)$ and $\underline{step\_ssat}(U,k)$ are lists of formulas; $step_2(U,k,l) = path(s_0,\ldots,s_{k+1}) \wedge P(s_0) \wedge \ldots \wedge P(s_k) \rightarrow P_l(s_{k+1})$, P_l is a property in $U = \{P_1,\ldots,P_n\}$, and $P = \bigwedge U$. Then two methods of performing simultaneous step are described by *STEP_SSAT* and *STEP_HYBRID* algorithms below:

```
STEP_SSAT(U,k) {
   (U,V) = SSAT*(step_ssat(U,k) )
   valid_callback(V);
   Return U;
}
```

Fig. 8. SSAT method for simultaneous induction step at depth *k*

Here we assume that $SSAT^*$ runs *SSAT* and returns a pair of lists, where the first list contains all properties P_j from U whose corresponding formulas $\neg step(P_j,k)$ get falsified by SSAT solver, and the second list consists of the remaining properties P_j from U; *valid_callback* reports all formulas P_j in V valid to the user. Indeed, for all such P_j from V, the corresponding step formula $step(P_j,k)$ is *unsatisfiable*, and since P_j was not falsified till depth k, it is *valid* according to the temporal induction scheme in [SSS00].

```
STEP_HYBRID (U,k) {
 V = U; // properties we may still prove valid at depth k
 U = ∅; // properties we already know cannot be proven at depth k
 fixpoint_reached = false;
 While ( ! fixpoint_reached ) {
       U_old = U;
       (U,V) = SSAT**( step_hybrid(U,k) )
       if ( U == U_old )
         fixpoint_reached = true;
  }
  valid_callback(V);
  Return U;
}
```

Fig. 9. Hybrid method for simultaneous induction step at depth k

Here $SSAT^{**}$ procedure runs *SSAT* and updates U and V as follows: it moves from V to U all formulas P_j whose corresponding step formulas $step_2(U,k,j) = path(s_0,\ldots,s_{k+1}) \wedge P(s_0) \wedge\ldots\wedge P(s_k) \rightarrow P_j(s_{k+1})$ from the list *step_hybrid*(U,k) get falsified in SSAT. When there are no such formulas, the while loop stops – *fixpoint_reached* is assigned *true*. Note that in such a case $path(s_0,\ldots,s_{k+1}) \wedge P(s_0) \wedge\ldots\wedge P(s_k) \wedge \neg P(s_{k+1})$ is *unsatisfiable*, and it is the step formula for the conjunct P, thus P (and all its conjuncts) are valid according to the temporal induction scheme in [SSS00]. The callback *valid_callback* reports all properties in V valid to the user.

Notice the differences between *STEP_SSAT* and *STEP_HYBRID* algorithms. *STEP_SSAT* needs one call to SSAT, thus in general is faster than *STEP_HYBRID* which may require more calls to SSAT. However, *STEP_HYBRID* works more like *STEP_CONJUNCTION* in that it can find the maximal subset of U whose conjunction can be proved at a given depth k. On the other hand, in *STEP_SSAT*, each unresolved property P_j is proved "without help of other properties", meaning that P_j at depth $k+1$ is attempted to prove based on assuming P_j valid at depths 0 to k, while *STEP_HYBRID* and *STEP_CONJUNCTION* use stronger assumptions that the conjunction of all unresolved properties in U is valid at depths 0 to k. Thus *STEP_HYBRID* and *STEP_CONJUNCTION* may in general prove more properties at a given depth than *STEP_SSAT*. The difference between *STEP_HYBRID* and *STEP_CONJUNCTION* is that the latter uses a SAT solver rather than SSAT; therefore the number of calls to the SAT solver depends on the returned models, and since in these models normally not all the properties falsifiable under the current step assumption come up *false*; in general *STEP_CONJUNCTION* needs much more iterations than *STEP_HYBRID*. This is one of the main advantages of *STEP_HYBRID* over *STEP_CONJUNCTION*.

Several variations of *simulatenous_base* and *simultaneous_step* are also possible. For example, one may choose to use the GN algorithm instead of SSAT in simultaneous base and step schemes proposed in this section. We have already mentioned that variations are possible in combining the base and step parts of induction, and increments to the depth other than 1 can easily be allowed by slight modification of the base and step formulas.

All base and step procedures described in this section can be made incremental in various ways. The conjunction method (and implementation) proposed in [FIK+02] is non-incremental, but it can easily be made incremental in the PISAT style of [WKS01, Str04] which needs tracking of pervasive learned clauses, or in the FISAT style of [ES03] where all learned clauses are pervasive. In the former approach, variables used to define the involved base and step formulas can be soundly removed from the instances, while for the latter option one needs to keep them. Since in SSAT all learned clauses are pervasive, it is safe to use them across all calls to SSAT solver at the same or different depths, as long as no variables and clauses used to define the involved base and step formulas are removed; indeed, since no temporal assumptions are made before SSAT calls, all learned clauses are logical consequences of the unrolled instances. From our experience, the extra defining clauses and variables of the base and step formulas are not a significant overhead to the SSAT solver.

6 Experimental Results

In this section, we report experimental results on some important applications in formal hardware verification domain where simultaneous and incremental SAT solving is very beneficial. In particular, we will compare the performance of the basic simultaneous induction algorithm when different simultaneous base and step procedures and different incremental schemes are used. All benchmarks originate from Intel designs. The performance results were generated on a 3.2 GHz machines with 4GB memory.

Since there are many variations of simultaneous temporal induction, there is a choice to be made here. As a base line, we choose the non-incremental conjunction method of [FIK+02]. We will refer to it as **conj**. We will consider a double-incremental version of it, in the PISAT style – we will refer to it as **dincr_conj**; this method is double incremental as we transfer the learned clauses from iteration to iteration at the same depth, as well as from depth to depth. It will allow us to measure the effect of pervasive learning on simultaneous induction. To measure the effect of fully incremental approach FISAT, we will consider SSAT-based schemes where the simultaneous falsification feature (which requires watching of all objectives) is disabled; we will consider two options: **incr_gn**, where learned clauses are not transferred from low to higher depths; and **dincr_gn**, where learned clauses are transferred to higher depths (note that the [GN01] approach corresponds to **incr_gn** rather than **dincr_gn**, since they did not consider incremental learning for related instances – rather, they considered the same instance for multiple objectives). We then consider the SSAT-based induction schemes – *BASE_SSAT* as *simultaneous_base* algorithm, and both *STEP_SSAT* and *STEP_HYBRID* as *simultaneous_step*. We will consider the double-incremental versions for both schemes, and refer to them as **dincr_ssat** and **dincr_hybr**, respectively. Furthermore, to measure the effect of double-incremental learning in SSAT-based induction as well, we will disable transferring the learned clauses from low to higher depths in **dincr_ssat**; we call the resulting scheme **incr_ssat**. We will not consider the scheme precisely corresponding to [ES03], as in the majority of our benchmarks we have tens or hundreds of properties in the same session, and even non-incremental simultaneous

induction schemes are superior to repeated application of incremental induction for single safety properties, when solving multiple properties.

The benchmark Tables 1-2 below originate from simultaneous SAT-based model checking of 543 invariant properties. The pruned model (that includes only the "cone of influence" of the properties) contains 2723 state elements (latches), 37159 logic gates and 3767 inputs. The first table gives data of a BMC run using both SSAT and PISAT algorithms at depths 6-15 (the lower depths took less than a second each to complete; the depth count in our algorithm starts from 0). And the second table gives similar data for the step part of the induction algorithm. The combinational instances at each depth are represented as $\wedge$ / $\neg$ graphs (and-inverter graphs, or AIGs, allow for a compact representation of Boolean formulas, see e.g. [KGP01]) and then are translated to the CNF representation to run PISAT or SSAT algorithm. The PISAT algorithm requires multiple calls to the DPLL algorithm, to resolve each (non-trivial) PO. For each PO, the corresponding cone of influence is built, and then translated to CNF representation. Usually, lots of optimizations are performed when translating an AIG representation into a CNF representation, allowing one to reduce significantly the amount of variables in the CNF instance. The downside is that performing such optimizations for each PO separately may be quite an overhead in some cases, especially when the "cones of influences" of the POs have a significant overlap. Usually, vectors of invariant properties are formed so that the properties share large chunks of common logic (otherwise there would be no reason for performing simultaneous model checking). This is the case in the benchmarks reported below.

Table 1. BMC at depths 6-15

BMC depth	# of POs	# of gates	# of inputs	# of literals	# of clauses	PISAT (sec)	GN (sec)	SSAT (sec)
6	543	98288	25383	93784	254145	174.36	9.28	2.42
7	494	113938	28885	108488	295157	245.51	5.41	4.68
8	473	132372	33352	125745	342993	210.47	8.14	3.96
9	450	150565	37454	142720	390432	316.93	2.61	2.61
10	450	170016	42072	160938	440968	305.79	11.79	6
11	435	189670	46529	179233	492157	508.97	14.61	11.6
12	418	209885	51380	198212	544750	364.76	7.68	6.69
13	417	229883	55880	216769	596763	576.3	5.72	5.71
14	417	250285	60745	235896	649809	424.04	11.14	11.38
15	415	270393	65243	254571	702148	686.75	7.96	8.05
Total						**3813.88**	**84.34**	**63.1**

Tables 1-2 show the size of the entire instance both in its AIG representation as well as in CNF representation. In these runs, around 40% of total runtime was spent on the pruning (i.e., relevant cone formation and CNF re-generation) part of the PISAT algorithm (the reported PISAT run times include the pruning times). And the PISAT algorithm spends more time in DPLL search than the SSAT algorithm

even if the pervasive learned conflicts are re-used in the PISAT algorithm (we know however that SSAT allows sharing of more conflict clauses). Overall, these tables demonstrate that the SSAT algorithm can perform orders of magnitude faster than the PISAT algorithm, at least on many practical designs. Furthermore, even if the time-consuming pruning is not performed in the GN algorithm, the benchmarks show that overall SSAT is significantly faster than GN.

Table 2. Induction step at depths 1-10

Step depth	# of POs	# of gates	# of inputs	# of literals	# of clauses	PISAT (sec)	GN (sec)	SSAT (sec)
1	543	52786	8026	45811	132490	87.42	30.3	1.99
2	433	73274	9024	62397	184397	117.71	44.6	2.2
3	433	94927	10065	79932	239277	170.86	62.13	3.26
4	433	117160	11128	97964	295687	230.31	78.97	3.92
5	433	140251	12196	117095	355143	291.87	97.73	5.03
6	384	161723	13135	134894	410561	323.46	109.5	15.93
7	260	182905	14182	152888	465867	273.91	117.62	10.36
8	236	204098	15210	170834	521100	292.09	117.87	8.56
9	236	225355	16241	188819	576470	321.43	131.28	11.03
10	221	246438	17261	206653	631386	340.79	139.34	9.24
Total						**2449.85**	**929.34**	**71.52**

Data in Tables 3-4 originate from compositional formal sequential equivalence verification runs [KSKH04]. The specification and implementation circuits are divided into corresponding sub-circuits and verified separately. Equivalence of the circuits can be derived from the equivalence of the corresponding sub-circuits. The properties that are checked state that the corresponding pairs of sub-circuit outputs in the specification and implementation models have same values for any input vector sequence of the respective sub-circuits.

We report experimental data on 6 vectors of safety properties. In order to show the overall impact on the run-time of the end tool (in this case, a SAT-based model checker), in this and other tables below we give total runtimes (BMC or Induction till a certain bound), and do not report a breakdown of times spent at each depth. Tables 3 and 4 both originate from the same model-checking runs (simultaneous induction).

In Table 3 we report the data showing the impact of several advanced features of the SSAT algorithm that do not exist in PISAT and FISAT. The POs in SSAT represent the base and step formulas at different depths. We report the amount of models discovered by SSAT and the number of POs falsified using these models – on average, each model was sufficient to falsify 8-9 POs. We also report the amount of POs proved valid based on global assignments before the end of search (i.e., based on a partial search) – all other POs were trivial to resolve, and they were not passed to the SAT solvers (their validity was discovered during translating the AIG representation to CNF). More interestingly, we give also data on the ratio of pervasive conflicts in the PISAT algorithm, and the amount of conflicts in the SSAT algorithm.

Table 3. Full proof with induction: conflicts, globally true, and simultaneous faslification data

# properties	#PISAT pervasive conflicts	#PISAT conflicts	#SSAT conflicts	#SSAT globally true	#SSAT models	#SSAT false	Bound reached
9	6170	35225	13615	126	21	189	22
9	8549	40680	15709	135	21	189	22
9	8258	37814	14206	135	21	189	22
9	8488	39945	14276	135	21	189	22
9	7056	35370	14251	135	21	189	22
9	6968	13257	13257	135	21	189	22

Table 4. Full proof induction

#properties	Conj	dincr-conj	dincr-gn	incr-ssat	dincr-ssat	dincr-hybr
9	189.57	72.67	66.65	35.05	29.84	29.32
9	200.13	73.23	61.72	40.46	27.76	29.09
9	222.29	66.11	67.17	35.84	26.06	27.83
9	246.51	67.85	62.33	37.22	28.24	29.5
9	253	68.14	59.55	39.04	28.66	30.13
9	215.09	70.25	60.5	35.52	26.7	27.86
Total (sec)	**1326.59**	**418.25**	**377.92**	**223.13**	**167.26**	**173.73**

Table 5. BMC run times (sec) till different depth

#properties	BMC depth	conj	dincr-conj	incr-gn	dincr-gn	incr-ssat	dincr-ssat
32	50	32.95	5.4	536.3	8.31	533.87	8.14
32	50	32.85	5.46	543.15	8.42	534.85	8.15
3	50	318.87	108.54	3041.72	46.17	3064.57	46.05
3	50	360.67	464.32	3760.78	210.15	3747.52	210.45
3	50	310.64	367.93	3653.52	50.23	3612.3	50.06
3	50	242.4	231.59	3337.25	199.96	3330.65	199.46
8	50	78.8	30.69	681.68	27.46	685.82	27.18
8	50	78.14	30.77	680.2	26.97	682.88	26.91
8	50	18.53	3.28	162.84	9.98	157.66	10.03
8	50	18.46	3.35	157.25	10.23	157.69	10.18
543	15	139.82	41.18	51.26	32.78	36.08	18.91
172	30	145.75	52.13	76.83	40.5	63.59	28.06
1035	3	2478.61	406.2	1743.56	1618.58	386.93	229.28
Total run times:		**4256.49**	**1750.84**	**18426.34**	**2289.74**	**16994.41**	**872.86**

There is a clear correlation between these conflict counts and the runtimes of the **dincr_conj** and **dincr_hybr** algorithms reported in Table 4. In Table 4, one can also see the advantage of the double-incremental verification. And furthermore, the SSAT-based algorithms **dincr-ssat** and **dincr-hybr** are the fastest among all simultaneous induction algorithms discussed in previous sections.

In Table 5 we report the model checking runtimes for various methods of simultaneous BMC on test cases originated from formal property verification as well as formal equivalence verification of Intel designs. Again, the SSAT-based algorithm **dincr_ssat** is clearly superior.

In Table 6, we compare the main double-incremental schemes, and show the speedup of **dincr_ssat** and **dincr_hybr** compared with **dincr_conj** (columns speedup 1 and speedup 2 respectively). The superiority of the SSAT based schemes is evident. We also present data on the number of properties, and number of properties proved valid or falsifiable. In the corresponding column, for the proved properties we give two figures: properties proved in **dincr_hybr** and properties proved in **dincr_ssat**; we already know that the former scheme may prove more properties than the latter.

Table 6. Full induction till bounds 10-30

#PO/#false/#valid	depth	conj	dincr_conj	dincr_ssat	dincr_hybr	speedup 1	speedup 2
172/106/65	30	2777.54	1718.51	200.69	237.66	8.56	7.23
543/128/(275:214)	30	3978.52	3287.89	312.08	233.97	10.54	14.05
100/0/(73:41)	10	18040.21	6390.95	830.61	1384.08	7.69	4.62
249/32/(70:64)	14	31338.58	7730.62	3710.35	2148.58	2.08	3.60

7 Conclusions

We presented an incremental propositional satisfiability technique allowing one to solve simultaneously and efficiently multiple satisfiability problems for related formulas. Insignificant modification to a (regular) DPLL-based SAT solver is sufficient to implement our Simultaneous SAT algorithm. Further, we presented several novel techniques for simultaneous SAT-based model checking of multiple safety properties. We provided experimental results demonstrating that the SSAT algorithm may be orders of magnitude faster in solving related SAT problems than the previous incremental SAT solving approaches that require multiple calls to a SAT solver, and that double-incremental simultaneous model checking of related safety properties employing the SSAT algorithm can accelerate verification significantly.

References

[BCCZ99] Biere A., A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1999.

[BCC+03] Biere, A., A. Cimatti, and E. Clarke, O. Strichman, Y. Zhu, Bounded Model Checking, Chapter in Advances in Computers, vol. 58, 2003.

[Bry86] Bryant R.E., Graph-based algorithms for Boolean function manipulation, IEEE Trans. Computers, C-35(8), 1986.

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

[DLL62] Davis M., G. Logemann, D. Loveland, A machine program for theorem proving. In Communications of the ACM, (5):394-397, 1962.

[DP60] Davis M., H. Putnam, A computing procedure for quantification theory, J. ACM, vol 7, 1960.

[ES03] Eén N, N. Sörensson, Temporal induction by incremental SAT solving, International Workshop on Bounded Model Checking, BMC 2003.

[FIK+02] Fraer, R., S. Ikram, G. Kamhi, T. Leonard, A. Mokkedem, Accelerated verification of RTL assertions based on satisfiability solvers, HLDVT, 2002.

[GN01] Goldberg E., Y. Novikov, An efficient learning procedure for multiple implication check. In Design, Automation, and Test in Europe (DATE '01), 2001.

[GSK98] Gomes C.*P*., B. Selman, H. Kautz, Boosting combinatorial search through randomization, National Conference on Artificial Intelligence, 1998.

[KGP01] Kuehlmann A., M.K. Ganai, *V*. Paruthi, Circuit-based Boolean reasoning, DAC 2001.

[KSKH04] Khasidashvili, Z., M. Skaba, D. Kaiss, Z. Hanna, Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints, ICCAD'04, 2004.

[LM02] Lynce I., J. Marques-Silva, Building state-of-the-art SAT solvers, European Conference on Artificial Intelligence (ECAI), 2002.

[MS97] Marques-Silva J.*P*., K.A. Sakallah, Robust search algorithm for test pattern generation, IEEE Fault-Tolerant Computing Symposium, 1997.

[MS99] Marques-Silva J.*P*., K.A. Sakallah, GRASP: A search algorithm for propositional satisfiability, IEEE Transactions on Computers, vol. 48, 1999.

[McM93] McMillan, K.L., *Symbolic Model Checking*, Kluwer, 1993.

[Nad02] Nadel A. Backtrack search algorithms for propositional satisfiability: Review and Innovations, Master Thesis, the Hebrew University of Jerusalem, 2002.

[PBG05] Prasad M., A. Biere, A. Gupta, A survey of recent advances in SAT-based formal verification, Int. Journal on Software Tools for Technology Transfer (STTT), vol. 7, number 2, 2005.

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

[Str04] Strichman, O., Accelerating bounded model checking of safety properties, Formal Methods in System Design, vol, 24, 2004.

[ZM88] Zabih R., D.A. McAllester, A rearrangement search strategy for determining propositional satisfiability, National Conference on Artificial Intelligence, 1988.

[ZMM+01] Zhang, L., C.F. Madigan, M.H. Moskewicz, S. Malik, Efficient conflict driven learning in a boolean satisfiability solver. International Conference on Computer-Aided Design (ICCAD'01), 2001.

[WKS01] Whittemore, J., K. Kim, K. Sakallah, SATIRE: A new incremental satisfiability engine, DAC, 2001.