2013 International Conference on Engineering of Complex Computer Systems

Reusing Search Tree for Incremental SAT Solving of Temporal Induction

Liangze Yin, Fei He, Min Zhou and Ming Gu
Key Laboratory for Information System Security, Ministry of Education
Tsinghua National Laboratory for Information Science and Technology
School of Software, Tsinghua University, Beijing 100084, PR China
Emails: yinliangze@163.com, {hefei, guming}@tsinghua.edu.cn, zhoumin03@mails.tsinghua.edu.cn

***Abstract*—Temporal induction is a SAT-based model checking technique. We prove that the SAT instances generated by its induction rule can be reduced to the so called Incremental CNFs. A new DPLL procedure is customized for Incremental CNFs, so that the intermediate results in solving previous instances, including the learnt clauses and the search tree, can be reused in solving the next instance. To the best of our knowledge, this is the first result on reusing the search tree in SAT solving of temporal induction. Experimental results on a large number of benchmarks show significant performance gain of our approach.**

***Keywords*—search tree; Incremental SAT; bounded model checking; temporal induction;**

I. Introduction

Bounded model checking (BMC) [1] [2] has made great success in both hardware and software verification. With the help of a SAT solver, it limits the search within a bounded length, and then reduces the model checking to a propositional satisfiability problem. Temporal induction [3] [4] [5] [6] is an enhancement to the bounded model checking. It can not only find the counterexamples, but can also prove the correctness.

Given a safety property, the temporal induction proof consists of two parts: the base case and the induction step. The base case states that the property should hold on all states that are reachable within k cycles from the initial states. The induction step states that the property should hold on the $(k+1)$-th state whenever it holds on k consecutive states. The length k is initialized to be 0, and increases by 1 each time, until either a counterexample is found or the property is verified.

With different values of k, the temporal induction proof gives off a series of SAT instances. Note that these SAT instances are generated by the same induction rule, and they are much similar in their structures. There are some works on exploiting the similarity of these SAT instances [6] [7] [8] [9] [10]. However, in all existing works, only learnt clauses are reused to help solving the next instance.

We prove in this paper that the SAT instances generated by the induction step can be reduced to the so called *Incremental CNFs*. With the Incremental CNFs, we observed that the counterexample (solution) for the instance of length k can possibly be extended (or is close to be extended) to the instance of length $k+1$, which in practice can reduce the running time of temporal induction on the designs with the large total time of intermediate induction queries. Based on this observation, we customize the DPLL procedure for Incremental CNFs so that the search tree of the previous instance can be reused. Given a new instance, the modified DPLL procedure starts the search from the last node of the former SAT solving, and then continues to traversal the search space. However, in each new iteration, some unit clauses might be added incrementally, which makes the search process backtrack to the 0-th decision level and consequently results in the whole search tree been cleared. To make maximal use of the previous search tree, a preprocessing step is proposed to eliminate unit clauses from the increments.

To the best of our knowledge, this is the first work on reusing the search tree in solving incremental SAT instances of temporal induction. The temporal induction by incremental SAT solving has been proposed in [6]. But their approach reuses the learnt clauses only. We implement our ideas in NuSMV, and compare it with the approaches in [6]. Experimental results on benchmarks of HWMCC'2011 show significant performance gain of our approach over theirs.

The rest of this paper is organized as follows. Section 2 briefly reviews temporal induction and SAT. Section 3 introduces our incremental SAT solving algorithm. Section 4 describes our method to reduce the temporal induction to incremental CNFs. The experimental results are reported in Section 5. Finally the paper is concluded in Section 6.

A. Related Works

Many research studies have been conducted on exploiting the similarity of the SAT problems in BMC. In [9], Shtrichman suggests adding replicated learnt clauses, some new clauses symmetric to the original learnt clauses. In [10], Wang et al. proposes to determine the variable decision order of the current problem by analyzing all previous unsatisfiable instances. Another successful technique is solving the sequence of problems incrementally by an incremental satisfiability engine [6] [7] [8]. This idea is based on the fact that most of the SAT solvers can learn learnt clauses, which act as a cache for the same type of conflicts in later search parts. The motivation for incremental temporal induction is that the learnt clauses are not only useful in later search parts of the same SAT problem, but can also be used in later similar problems.

An incremental SAT solver is proposed in [11], which reuses the search tree in solving a series of SAT instances. Their idea for incremental SAT solving is similar to ours.

978-0-7695-5007-7/13 $26.00 © 2013 IEEE
DOI 10.1109/ICECCS.2013.21

However, we further utilize the incremental SAT solving to perform temporal induction. Additionally, their approach requires that the new coming SAT instance has exactly one more clause than the former instance. Our approach loosens this constraint and the increment can contain a set of clauses. Consequently, when a new clause is added, [11] tries to find a solution as soon as possible, while our algorithm backtracks to the first point making the clause unit if it is a unit or conflict clause. Our approach is more efficient in cases where a large number of clauses need to be added.

Another work related to ours is the random restarts [12] [13], which has also been considered as a successful improvements in modern SAT solvers. It frequently clears the search tree and backtracks to the top level to avoid the "heavy tails" phenomenon [13]. On the surface, our work is contrary to this technique, for that we try to reuse the search tree while it clears the search tree. But in fact, our work can be combined with it effectively. Our motivation is to extend the counterexample of the previous instance to the next instance. If the counterexample can be extended, a solution for the next instance can be found quickly; otherwise it will leads to the "heavy tails" phenomenon, and the restarts technique will clear the search tree and restart the search process.

II. Preliminaries

A. Temporal Induction

A Finite State Machine (FSM) is a triple $M(\mathbf{x}, I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'))$, where $\mathbf{x}$ is the set of variables, $I(\mathbf{x})$ is the initial condition predicate and $T(\mathbf{x}, \mathbf{x}')$ is the transition relation predicate. Throughout the paper, we assume $|\mathbf{x}| = n$, and let $\mathbf{x} = \{x^0, x^1, \ldots, x^{n-1}\}$. A valuation of $\mathbf{x}$ gives a state of M. We use subscripts to denote the cycles, with $\mathbf{x}_i$ $(i \geq 0)$ being a copy of $\mathbf{x}$, representing the state variables at the i-th cycle. In this paper, the computation tree logic (CTL) [14] [15] is used to specify the properties. Given a propositional formula $P(\mathbf{x})$, $\mathbf{AG}P(\mathbf{x})$ denotes a safety property on M, which states that $P(\mathbf{x})$ holds at every reachable state of M. Another operator $\mathbf{AX}$ specifies that a property holds in the second state of all the path. We can write $\mathbf{AG}P(\mathbf{x}) = P(\mathbf{x}) \wedge \mathbf{AXAG}P(\mathbf{x})$. Temporal induction focuses on the safety property verification on FSMs. In the following, we do not distinguish $P(\mathbf{x})$ from $\mathbf{AG}P(\mathbf{x})$. We use I_i, T_i, P_i as a shorthand to denote $I(\mathbf{x}_i)$, $T(\mathbf{x}_i, \mathbf{x}_{i+1})$, and $P(\mathbf{x}_i)$, respectively. The temporal induction with length k consists of two steps: $Base_k$ and $Induct_k$, which are formulated as:

$$\begin{aligned} Base_k &= I_0 \wedge (\textstyle\bigwedge_{i=0}^{k-1}(T_i \wedge P_i)) \wedge (\neg P_k) \\ Induct_k &= (\textstyle\bigwedge_{i=0}^{k}(T_i \wedge P_i)) \wedge (\neg P_{k+1}) \wedge Unique_{k+1} \\ Unique_k &= (\textstyle\bigwedge_{0 \leq i < j \leq k} \mathbf{x}_i \neq \mathbf{x}_j) \\ &= (\textstyle\bigwedge_{0 \leq i < j \leq k}(\bigvee_{t=0}^{n-1}(x_i^t \neq x_j^t))) \end{aligned} \tag{1}$$

The basic temporal induction algorithm is demonstrated in Algorithm 1, which has been proved sound and complete [6]. In this algorithm, $solve(f)$ computes the satisfiability of formula f, and length k keeps increasing monotonically until either $Base_k$ becomes satisfiable (a counterexample is found) or $Induct_k$ turns unsatisfiable (the property is proved to be true).

Algorithm 1 Een-Sorensson

Input: InitState I_0, Transition T, Property P

```
1: for k = 0 to ∞ do
2:    if solve(Base_k) == SAT then
3:       return Property P is false
4:    end if
5:    if solve(Induct_k) == UNSAT then
6:       return Property P is true
7:    end if
8: end for
```

B. SAT Solving

Most modern SAT solvers take conjunctive normal form (CNF) as input. Given a propositional variable x, a literal is either the positive- or negative-form of x. Given a literal l, $var(l)$ is used to denote the variable of l. A CNF is a conjunction of clauses, where each clause is a disjunction of literals. In this paper, a CNF $\mathbf{f}$ is represented by a tuple $\langle X_{\mathbf{f}}, CL_{\mathbf{f}} \rangle$, which denote the variable set and clause set of $\mathbf{f}$ respectively. Given a CNF $\mathbf{f}$, an assignment τ is a valuation of $X_{\mathbf{f}}$. Specially, if just a part of the variables in $X_{\mathbf{f}}$ are assigned, then τ is called a partial assignment. For each variable $x \in X_{\mathbf{f}}$, $\tau(x)$ is used to denote the value of x in τ. Given τ, an assignment (or a partial assignment) of CNF $\mathbf{f}$, and a clause $C \in CL_{\mathbf{f}}$, τ conflicts C iff C is false in τ; otherwise, if C is true in τ, then we say that τ satisfies C; τ conflicts $\mathbf{f}$ iff $\exists C \in CL_{\mathbf{f}}$ so that τ conflicts C; τ satisfies $\mathbf{f}$ iff $\forall C \in CL_{\mathbf{f}}$, τ satisfies C. Any propositional formula f can be converted into CNF $\mathbf{f}$ in linear time by importing intermediate variables [16] [17] [18] [19]. In most of the CNF conversion algorithms, a new name is given to every subformula of f, which is known as the *definitional* clause, and the resulting CNF is *equisatisfiable* to the original formula f. For example, formula $f = ((a \wedge b) \vee c) \wedge d$ can be represented by formula set $S = \{x \Leftrightarrow a \wedge b, y \Leftrightarrow x \vee c, z \Leftrightarrow y \wedge d, z\}$ where each subformula $t \in S$ can be converted to clauses trivially. Here $\{x, y, z\}$ is the imported intermediate variable set, and z, the new name of formula f, is called the formula literal of $\mathbf{f}$.

Most modern SAT solvers are based on the DPLL algorithm [20] [21]. It is a Depth-First-Search (DFS) algorithm on exploration of the variable space, which implicitly transverse the search space of 2^n possible assignments to the problem variables. The paths traversed by the DFS algorithm establish a tree gradually, which is called the search tree. At each node of the search tree, the DPLL algorithm selects a variable and assigns it true or false; this process is called making a *decision*. The depth when a variable x is assigned is called the decision level of x. A clause is unit iff it has only one free (unassigned) literal and all the other literals are false. The free literal of a unit clause must be set true to make the clause satisfiable; this is called the Unit Clause Rule. The process of applying the Unit Clause Rule iteratively is called Boolean Constrain Propagation (BCP). A clause is a conflict clause iff all its literals are false. A conflict is encountered in the search process iff some conflict clause is found. The DPLL algorithm iteratively makes decisions and applies BCP, until either a satisfiable assignment is found or a conflict is encountered.

If it is the latter case, it backtracks and makes a new decision unless the search space is fully explored, which indicates that the CNF is unsatisfiable. In the last decades, clause learning has been substantiated as the most popular technique in modern SAT solvers. It is based on the idea that whenever a conflict is detected during the search process, it analyzes the reason for the conflict that can be encoded as a clause (the learnt clause). All the learnt clauses are managed in a learnt clause database, which acts as a cache to prune the search space in the later parts.

In the search tree of the DPLL algorithm, every path from the root to a leaf node is an assignment (or a partial assignment) of the CNF. In modern SAT solvers, these paths are implicitly represented by the variable assignments when a conflict is encountered or the search process ends. If the problem is satisfiable, then all the assignments corresponding to these paths conflict the CNF, except the one from the root to the last node examined.

III. Incremental SAT Solving with Search Tree Reused

Definition 1 ($\mathbf{f}_1 < \mathbf{f}_2$): For two CNFs $\mathbf{f}_1, \mathbf{f}_2$, $\mathbf{f}_1 < \mathbf{f}_2$ iff $CL_{\mathbf{f}_1} \subset CL_{\mathbf{f}_2}$ and $X_{\mathbf{f}_1} \subseteq X_{\mathbf{f}_2}$.

Let $Inc_{\mathbf{f}_1,\mathbf{f}_2} = CL_{\mathbf{f}_2} \setminus CL_{\mathbf{f}_1}$. Obviously, $Inc_{\mathbf{f}_1,\mathbf{f}_2} \neq \emptyset$ if $\mathbf{f}_1 < \mathbf{f}_2$.

Theorem 1: Given two CNFs $\mathbf{f}_1, \mathbf{f}_2$ where $\mathbf{f}_1 < \mathbf{f}_2$, for any partial assignment τ of $\mathbf{f}_1$, if τ conflicts with $\mathbf{f}_1$, then τ conflicts with $\mathbf{f}_2$.

Proof: Since τ conflicts $\mathbf{f}_1$, there must exist a clause $C \in CL_{\mathbf{f}_1}$ so that C is false in τ. According to the definition of $\mathbf{f}_1 < \mathbf{f}_2$, $CL_{\mathbf{f}_1} \subset CL_{\mathbf{f}_2}$, then $C \in CL_{\mathbf{f}_2}$. Thus τ conflicts with $\mathbf{f}_2$. ■

Given two CNFs $\mathbf{f}_1$, $\mathbf{f}_2$ where $\mathbf{f}_1 < \mathbf{f}_2$, Theorem 1 gives us the possibility to conclude $\mathbf{f}_2$'s unsatisfiability from $\mathbf{f}_1$'s unsatisfiability. The reverse does not hold. If $\mathbf{f}_1$ is satisfiable, we cannot directly decide whether $\mathbf{f}_2$ is satisfiable or not.

Theorem 2: Given two CNFs $\mathbf{f}_1, \mathbf{f}_2$ where $\mathbf{f}_1 < \mathbf{f}_2$, suppose $\mathbf{f}_1$ is satisfiable, Ψ is the set of assignments corresponding to the paths in $\mathbf{f}_1$'s search tree which start from the root and end at a leaf node, and τ_0 is the assignment corresponding to the path from the root to the last node examined, then $\forall \tau \in \{\Psi \setminus \tau_0\}$, τ conflicts with $\mathbf{f}_2$.

Proof: Since $\mathbf{f}_1$ is satisfiable, $\forall \tau \in \{\Psi \setminus \tau_0\}$, τ conflicts with $\mathbf{f}_1$. According to Theorem 1, $\forall \tau \in \{\Psi \setminus \tau_0\}$, τ conflicts with $\mathbf{f}_2$. ■

From Theorem 2, if $\mathbf{f}_1$ is satisfiable and we are going to determine $\mathbf{f}_2$'s satisfiability, then the search tree of $\mathbf{f}_1$ can be reused. As the search tree does not explicitly exists in modern SAT solvers, reusing the search tree here means that there is no need to look again at any path that has already been traversed in solving $\mathbf{f}_1$, except the path from the root to the last node examined. So it is possible for us to keep all middle results in solving $\mathbf{f}_1$. In addition to conflict clauses (used in the previous work), the current state of the decision stack and all the propagations, etc, can also be reused. To solve $\mathbf{f}_2$, we simply add $Inc_{\mathbf{f}_1,\mathbf{f}_2}$ to the solver, and continue the search process from the last node examined.

Definition 2 (Incremental CNFs): Given a set of CNFs $\mathbf{F} = \{\mathbf{f}_1, \mathbf{f}_2, \cdots, \mathbf{f}_n\}$, $\mathbf{F}$ is a set of incremental CNFs iff $\forall i (1 \leq i < n), \mathbf{f}_i < \mathbf{f}_{i+1}$.

If a series of incremental CNFs $\mathbf{F} = \{\mathbf{f}_1, \mathbf{f}_2, \cdots, \mathbf{f}_n\}$ is to be solved sequentially, then for any $i < n$, we can continue the search process of $\mathbf{f}_i$ to solve $\mathbf{f}_{i+1}$. Algorithm 2 outlines our algorithm Incremental-Solving for incremental CNFs $\mathbf{F} = \{\mathbf{f}_1, \mathbf{f}_2, \cdots, \mathbf{f}_n\}$, which supports reusing the search tree of $\mathbf{f}_i$ to speed up the solving of $\mathbf{f}_{i+1}$ $(i < n)$. It returns false (Line 11) if all the CNFs are satisfiable, and returns true (Line 8) if there is an index i $(1 \leq i \leq n)$ such that $\forall j (j \geq i)$ $\mathbf{f}_j$ is unsatisfiable, and $\forall j (j < i)$ $\mathbf{f}_j$ is satisfiable.

Algorithm 2 Incremental-Solving

```
Input: Incremental CNFs F = {f_1, f_2, ..., f_n}
 1: s = new satSolver()
 2: s.addClause(CL_{f_1})
 3: for i = 1 to n do
 4:    ret = s.incSolve()
 5:    if ret == SAT then
 6:       s.furtherAddClauses(Inc_{f_i,f_{i+1}})
 7:    else
 8:       return true
 9:    end if
10: end for
11: return false
```

Note that from the beginning to the end, the Incremental-Solving algorithm has only one solver instance s. Function $addClause$ adds a set of clauses to the solver instance when the search tree is empty, while $furtherAddClauses$ adds clauses after the solver instance starts its search process. Function $incSolve$ continues the search process of the solver instance. When $incSolve$ returns, it does not clear the search tree or any clauses learnt in the solving process of previous CNFs. The $incSolve()$ function returns true if the current problem is satisfiable, and false otherwise.

A. Modified DPLL Procedure

This section customizes the DPLL procedure to support search tree reuse. Note this technique can easily be applied to any modern SAT solver if only it is based on the DPLL procedure.

From Algorithm 2, to conform our Incremental-Solving algorithm, the SAT solver should have two special functions: $incSolve$ and $furtherAddClauses$. Function $incSolve$ continues its search from the last node of the previous solving, which requires the SAT solver to keep all its data structures after solving one problem, including its search tree, learnt clauses, and variable assignments, etc. Function $furtherAddClauses$ provides the scheme of adding some new clauses to a solver instance that has already started its search process. As it is believed in modern SAT solvers that the earlier a unit clause is found, the more search space it may prune. So if some clause C is found to be unit when added to the solver, we backtrack to the first point making it unit, and assign the unassigned literal true. If C is found to be a conflict clause, as it should be found to be unit as soon as there is only one unassigned literal, we backtrack to that point and assign the unassigned literal true.

Algorithm 3 outlines the procedure for $furtherAddClauses$ function. In this algorithm, C is a clause with no duplicate literals, and $|C|$ is the number of literals in C. Function $backtrack(lv)$ backtracks to level lv; $assign(lit)$ assigns literal lit true; $clauseAdd(C)$ adds clause C to the original clause set. It returns true (Line 20) if C is added correctly, and returns false (Line 2, 7) if the problem in the solver instance is $UNSAT$. The crucial function is $UnitPointAnalysis$ (Line 5), which analyzes the first point making C unit if C is a unit or conflict clause. If C is a unit clause when adding to the solver instance, let l be its last literal to be assigned false, then C should become unit as soon as l is assigned, thus the decision level of l is the first point making C unit. If C is a conflict clause, then it should be unit when the literal with the second maximal decision level is assigned, and the literal with the maximal decision level is the unassigned literal at that point. Specially, if the two maximal decision level literals have the same decision level lv, then we backtrack to $lv-1$, and there are two unassigned literals at that point. The procedure for $UnitPointAnalysis$ function is shown in Algorithm 4. Here $var(l)$ is the variable of literal l; $level(x)$ is the decision level of variable x; l_1 and l_2 are the two literals of C with the maximal and the second maximal decision level respectively.

Algorithm 3 Further-Add-Clause

Input: Solver s, Clause C

```
1: if |C| == 0 then
2:    return false
3: else
4:    if C is a unit or conflict clause then
5:       lv = UnitPointAnalysis(s, C)
6:       if lv < 0 then
7:          return false
8:       else
9:          s.backtrack(lv)
10:         s.clauseAdd(C)
11:         if C is a unit clause then
12:            let l be the unassigned literal
13:            s.assign(l)
14:         end if
15:      end if
16:   else
17:      s.clauseAdd(C)
18:   end if
19: end if
20: return true
```

IV. Reduce Induction Step to Incremental CNFs

The SAT instances generated by the the temporal induction are highly related. This section focuses on the problems in the induction step, and tries to convert them into incremental CNFs by variable substitution, then they can be solved by our Incremental-Solving algorithm.

In the following, we let

$$\theta_k = (\bigwedge_{i=0}^{k}(T_i \wedge P_i)) \wedge (\neg P_{k+1})$$

$$\psi_k = (\neg P_0) \wedge (\bigwedge_{i=0}^{k}(T'_i) \wedge P_{i+1}))$$

Algorithm 4 Unit-Point-Analysis

Input: Solver s, Clause C

```
1: assert(C is a unit or conflict clause)
2: if C is a unit clause then
3:    if |C| == 1 then
4:       return 0
5:    else
6:       return level(var(l_1))
7:    end if
8: else if C is a conflict clause then
9:    if level(var(l_1)) > level(var(l_2)) then
10:      return level(var(l_2))
11:   else
12:      assert(level(var(l_1)) == level(var(l_2)))
13:      return level(var(l_2)) - 1
14:   end if
15: else
16:    return Error
17: end if
```

where $T'_i = T(X_{i+1}, X_i)$.

Theorem 3: The satisfiability of θ_k and that of ψ_k are equivalent.

Proof: Suppose $\mathbf{x} = \{x^0, x^1, \ldots, x^{n-1}\}$ is the variable set of the model. Consider formula θ_k, if we substitute each variable x_i^t ($0 \le t < n$ and $0 \le i \le k+1$) in θ_k with x_{k+1-i}^t, then we can get ψ_k. As we didn't change the structure of the formula, if there exists a solution τ for θ_k, then τ' must be a solution of ψ_k, where $\tau(x_i^t) = \tau'(x_{k+1-i}^t)$ ($0 \le t < n$, $0 \le i \le k+1$). The converse is also true, i.e., if there exists a solution τ' for ψ_k, then τ must be the solution of θ_k.

Thus the satisfiability of ψ_k and θ_k is equivalent. ■

Theorem 4: The satisfiability of $\theta_k \wedge Unique_{k+1}$ and that of $\psi_k \wedge Unique_{k+1}$ are equivalent.

Proof: We first prove that if $\theta_k \wedge Unique_{k+1}$ is satisfiable then $\psi_k \wedge Unique_{k+1}$ is satisfiable too. Suppose τ is one of the solutions of $\theta_k \wedge Unique_{k+1}$, let τ' be the assignment where $\tau(x_i^t) = \tau'(x_{k+1-i}^t)$ ($0 \le t < n$, $0 \le i \le k+1$). As τ satisfies θ_k, according to the analysis in the proof of Theorem 3, τ' satisfies ψ_k. As τ satisfies $Unique_{k+1}$, according to the symmetrical structure of τ and τ', τ' satisfies $Unique_{k+1}$ too. Thus τ' is a solution of $\psi_k \wedge Unique_{k+1}$, and $\psi_k \wedge Unique_{k+1}$ is satisfiable.

Similarly, if $\psi_k \wedge Unique_{k+1}$ is satisfiable then we can prove that $\theta_k \wedge Unique_{k+1}$ is satisfiable too.

Therefore, the satisfiability of $\theta_k \wedge Unique_{k+1}$ and that of $\psi_k \wedge Unique_{k+1}$ is equivalent. ■

Based on Theorem 4, instead of checking the satisfiability of $Induct_k = \theta_k \wedge Unique_{k+1}$, we can check the satisfiability of $Induct'_k = \psi_k \wedge Unique_{k+1}$. This approach is also mentioned in [6], which is called "growing the trace backwards" in its Dual algorithm. It is easy to see that $Induct'_k = Induct'_{k-1} \wedge F_{(i-1,i)}$ where $F_{(i-1,i)} = (P_{i+1} \wedge T'_i) \wedge (\bigwedge_{0 \le i < k+1}(\mathbf{x}_i \neq \mathbf{x}_{k+1}))$. After converting the problems of $Induct'_k$ ($k \ge 0$) into CNFs, it is easy to make them to

be incremental CNFs, and then they can be solved by our Incremental-Solving algorithm.

A. Incremental Clause Set Preprocess

The notable feature of our algorithm is that it can reuse the search tree of the previous problem. But according to Further-Add-Clause algorithm in Algorithm 3, if there exists some clause C so that $UnitPointAnalysis(s, C) == 0$, then the SAT solver backtracks to the top level, and clears the current search tree.

According to Algorithm 4, $UnitPointAnalysis$ returns 0 if C is a unit clause at the top level. It can be divided into two cases:

1) C is a one-literal-clause ($|C| == 1$).
2) $|C| > 1$, but at the top level it has only one unassigned literal, and all the other literals are false.

These cases are familiar in real problems. For example, for any $i \geq 1$, when adding clauses in $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$ to the solver instance, the definitional clause of the increment is just in the first case, which contains only the formula literal of the increment. This section tries to avoid these cases according to the characteristics of the temporal induction, so that the search tree can be reused to a certain extent. Our idea is to eliminate these clauses from the increments before they are added to the solver instance, in precondition of keeping the satisfiability of $Induct'_j$ $(j \geq i)$.

Definition 3 (Literal Elimination): Given a CNF $\mathbf{f} = \langle X_\mathbf{f}, CL_\mathbf{f} \rangle$ and a literal lit, eliminating lit from $\mathbf{f}$ means: $\forall C \in CL_\mathbf{f}$, if C contains $\neg lit$ then remove $\neg lit$ from C; otherwise if C contains lit, then remove C from $CL_\mathbf{f}$.

$\forall C \in CL_\mathbf{f}$ where $C = \{lit\}$, eliminating lit from $\mathbf{f}$ does not change the satisfiability of $\mathbf{f}$. Let $\mathbf{f}_i$ denote the CNF of $Induct'_i$. To solve $\{Induct'_0, Induct'_1, \cdots, Induct'_n\}$ sequentially, $\{CL_{\mathbf{f}_0}, Inc_{\mathbf{f}_0,\mathbf{f}_1}, \cdots, Inc_{\mathbf{f}_{n-1},\mathbf{f}_n}\}$ are added to the solver instance orderly in the Incremental-Solving algorithm. Obviously $CL_{\mathbf{f}_i} = \{CL_{\mathbf{f}_0}, Inc_{\mathbf{f}_0,\mathbf{f}_1}, \cdots, Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}\}$. Suppose we are going to add $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$, where $\mathbf{x}_i$ is the set of state variables in cycle i, and $\mathbf{w}_i$ is the set of imported medium variables when converting $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i} = (P_{i+1} \wedge T'_i) \wedge (\bigwedge_{0 \leq i < k}(S_i \neq S_k))$ to CNF, the following discusses how to eliminate unit clauses in these two cases respectively:

1) $|C| == 1$:

Theorem 5: Suppose $C = \{lit\}$ is a one-literal-clause in $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$, and $v = var(lit)$, then either $v \in \mathbf{x}_i$ or $v \in \mathbf{x}_{i+1}$ or $v \in \mathbf{w}_i$.

Proof: As $(\bigwedge_{0 \leq i < k+1}(\mathbf{x}_i \neq \mathbf{x}_{k+1}))$ does not imply any unit clause, C is deduced by $(P_{i+1} \wedge T'_i)$, which contains variables only in $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$. Thus $(v \in \mathbf{x}_i)$ or $(v \in \mathbf{x}_{i+1})$ or $(v \in \mathbf{w}_i)$. ■

In the following we discuss how to eliminate lit in the 3 cases listed in Theorem 5.

Theorem 6: If $v \in \mathbf{w}_i$, then eliminating lit from $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$ does not change the satisfiability of $\mathbf{f}_j$ $(j > i)$.

Proof: Since $v \in \mathbf{w}_i$, v exists only in $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$. So, to CNF $\mathbf{f}_j$ $(j > i)$, eliminating lit from $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$ is equivalent with eliminating lit from $CL_{\mathbf{f}_j}$. Thus it does not change the satisfiability of $\mathbf{f}_j$ $(j > i)$. ■

Theorem 7: If $v \in \mathbf{x}_{i+1}$, then eliminating lit from all $Inc_{\mathbf{f}_{k-1},\mathbf{f}_k}$ $(i \leq k \leq j)$ does not change the satisfiability of $\mathbf{f}_j$ $(j > i)$.

Proof: Since $v \in \mathbf{x}_{i+1}$, then for $CL_{\mathbf{f}_j}$ $(j > i)$, v exists only in $\{Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}, \cdots, Inc_{\mathbf{f}_{j-1},\mathbf{f}_j}\}$. So, to CNF $\mathbf{f}_j$, eliminating lit from all $Inc_{\mathbf{f}_{k-1},\mathbf{f}_k}$ $(i \leq k \leq j)$ is equivalent with eliminating lit from $CL_{\mathbf{f}_j}$. Thus it does not change the satisfiability of $\mathbf{f}_j$ $(j > i)$. ■

If $v \in \mathbf{x}_i$, then v may exists in $Inc_{\mathbf{f}_{i-2},\mathbf{f}_{i-1}}$, which has already been added to the solver instance, so lit cannot be eliminated. We first analyze that in which situation this case happens. Without loss of generality, we suppose that $v = x_i^t$ and $lit = x_i^t$.

Theorem 8: If $v \in \mathbf{x}_i$ then $P_i \wedge T_i \Rightarrow x_{i+1}^t$.

Proof: As $(\bigwedge_{0 \leq i < k+1}(\mathbf{x}_i \neq \mathbf{x}_{k+1}))$ does not imply any unit clause, then $(P_{i+1} \wedge T'_i) \Rightarrow x_i^t$. Swap $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$, then $P_i \wedge T_i \Rightarrow x_{i+1}^t$. ■

When we consider the satisfiability of $Induct'_k$, as $Base_j$ $(j \leq k)$ must be unsatisfiable, P_j $(j \leq k)$ is always true. According to Theorem 8, if $v \in \mathbf{x}_i$, then x_j^t $(j \leq k+1)$ is always true except x_0^t.

Suppose the model is $M = \{\mathbf{x}, I(\mathbf{x}_0), T(\mathbf{x}_i, \mathbf{x}_{i+1})\}$, to avoid that there exists an one-literal-clause $(C = \{lit\}) \in Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$ where $v = var(list)$ and $v \in \mathbf{x}_i$, we verify $P(\mathbf{x}) \wedge \mathbf{AXAG}P(\mathbf{x})$ instead of $\mathbf{AG}P(\mathbf{x})$. To verify $\mathbf{AXAG}P(\mathbf{x})$, we construct a new model M^* which sets the second state of M as its initial state. In M^*, the literals x_j^t $(j \geq 0)$ are all true, so the corresponding literal can be eliminated from the transition relation $T(\mathbf{x}_i, \mathbf{x}_{i+1})$. The new model $M^* = \{X, I^*(\mathbf{x}_1), T^*(\mathbf{x}_i, \mathbf{x}_{i+1})\}$ can be formulated as (2), where $\mathbf{L}$ is the literal set which can be deduced by $P_i \wedge T_i$, and $T(\mathbf{x}_i, \mathbf{x}_{i+1})|_\mathbf{L}$ is the formula obtained by eliminating all the literals in $\mathbf{L}$ from $T(\mathbf{x}_i, \mathbf{x}_{i+1})$.

$$\begin{cases} I^*(\mathbf{x}_1) & = I(\mathbf{x}_0) \wedge T(\mathbf{x}_0, \mathbf{x}_1) \\ T^*(\mathbf{x}_i, \mathbf{x}_{i+1}) & = T(\mathbf{x}_i, \mathbf{x}_{i+1})|_\mathbf{L}, i \geq 1 \end{cases} \tag{2}$$

2) $|C| > 1$: Suppose lit is the unassigned literal of C and $v = var(lit)$, then either $v \in \mathbf{w}_i$ or $v \in \mathbf{x}_{i+1}$ (otherwise lit should be assigned true instead of unassigned). So lit can be eliminated just as discussed in Theorem 6 and 7. The difference is that to determine which clause is a unit clause at the top level, we should know the variable assignments at the top level. Another issue is that when a literal is eliminated, it may imply other unit clauses, i.e., it is a propagation process, and all the propagated literals have to be eliminated.

V. Experimental Results

Zigzag and *Dual* [6] are two state-of-the-art incremental algorithms for temporal induction. They reuse learnt clauses only and have already been realized in NuSMV. Our algorithm (named "$IncUST$") is implemented on the top of

Dual algorithm of NuSMV 2.5.3[1]. In *IncUST*, the problems in base step are solved similarly as in *Dual*, and the problems in induction step are solved by our algorithm. We compare *IncUST* to both *Zigzag* and *Dual*. The SAT solver is minisat2-v070721[2]. To implement our idea, the function *incSolve* and *furtherAddClauses* discussed in Section III are added to minisat. All experiments are conducted on a computer with Intel(R) Core(TM) i5 CPU 2.67GHz with 4GB memory.

The test cases are taken from the main single safety/bad-state property track (called single track for short) of HWMCC (Hardware Model Checking Competition) 2011[3]. The single track has 467 instances in total, including 67 instances from intel which are listed as symbolic links and have to be obtained separately due to license restrictions, leaving 400 instances available. All of these instances are in new AIGER 1.9 format. We utilize the aiger-1.9.3[4] to translate the AIG file to SMV format.

In our experiments, the time limit is set as 900 seconds. Among the 400 instances, *Zigzag* algorithm can solve only 153 instances, including 110 trivial instances which can be solved in less than one second by all algorithms. We take the remaining 43 instances to compare *IncUST* to *Zigzag* and *Dual*.

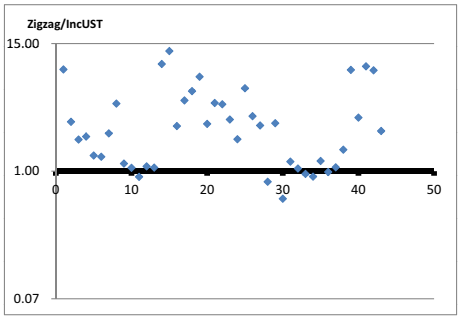
The experimental results are listed in TABLE I, where "Benchmark" represents the instance name, "# Step" is the number of cycles to solve this instance, "# Result" indicates the verification result, and "Type" is a flag we used for analyzing the results, which will be discussed later. Since the search tree reusing works only for the induction step, we compare both the time spent on induction step and the total runtime among these three algorithms, which are listed in "Induct" and "Total" columns respectively. The bold values indicate the best results among these algorithms.

From the table we can see that, *IncUST* runs the fastest for most of the instances. For induction time, there are 35 instances *IncUST* runs the fastest, 7 instances *Dual* runs the fastest, and 1 instances *Zigzag* runs the fastest. For total time, there are 29 instances *IncUST* runs the fastest, 8 instances *Dual* runs the fastest and 6 instances *Zigzag* runs the fastest. The following compares our algorithm with *Zigzag* and *Dual* algorithm respectively in detail.

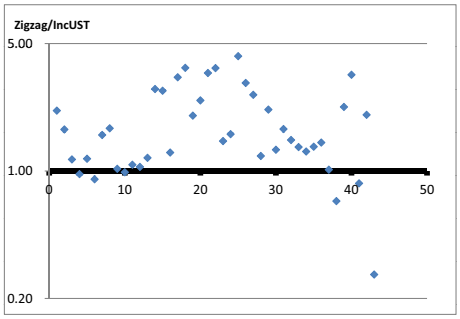
A. IncUST vs. Zigzag

Zigzag algorithm has only one SAT instance for base and induction step together, so learnt clauses of the two steps can be shared, which may bring some benefits for both the base and induction step. But for two instances algorithm (*Dual* and *IncUST*), those learnt clauses related with I_0 in base step and $\neg P_{k+1}$ in induction step can also be reused. In this sense, two instances algorithm is preferable.

The comparison of *IncUST* and *Zigzag* algorithm is illustrated in Fig. 1, where 1(a) compares the induction time and 1(b) compares the total time. In these two figures, the x-axis is the instances ID in TABLE I, while the y-axis indicates



(a) induction time



(b) total time

Fig. 1. Comparison between *IncUST* and *Zigzag*

the ratio of *Zigzag*'s time and *IncUST*'s time. The bold line in the two figures represents *Zigzag*'s time equals *IncUST*'s time. And for each instance, if the corresponding point is above the bold line then *IncUST* runs faster, otherwise *Zigzag* runs faster. The distance between the point and the bold line represents the ratio degree of the two algorithms. Fig. 2 in next section is the same. From the figure we can see that *IncUST* algorithm runs faster for almost all the instance in induction step. In the best case, it even runs 15 times faster. For the total time, *IncUST* also runs faster for most instances. For some special instances, as the base step of *Zigzag* may be much faster than *IncUST* algorithm due to its sharing of learnt clause between base and induction step, *Zigzag* wins over our *IncUST* algorithm in total time.

B. IncUST vs. Dual

The only difference between *IncUST* and *Dual* algorithm is that *IncUST* can reuse the search tree of $Induct'_k$ to help solving $Induct'_{k+1}$. Specially, when adding clauses, *Dual* uses the *addClause* function in minisat while *IncUST* uses the *furtherAddClauses* function; when solving the problem, *Dual* uses the *solve* function in minisat while *IncUST* uses the *incSolve* function; and in *IncUST* algorithm, all the increments will be preprocessed as discussed in Section IV-A. Fig. 2 demonstrates the comparison of *IncUST* and *Dual* algorithm. From TABLE I and Fig. 2 we can see that *IncUST* runs faster for most instances in both induction and total time. In fact, as the approaches of the two algorithms in base step are the same, the total time has the same distribution with induction time, except the ratio is reduced. But there are still some instances on which *IncUST* gets only a little improvement or even worse. To analyze them in detail, we

[1]http://nusmv.fbk.eu/
[2]http://minisat.se/MiniSat.html
[3]http://fmv.jku.at/hwmcc11/benchmarks.html
[4]http://fmv.jku.at/aiger/

TABLE I. EXPERIMENTAL RESULTS ON BENCHMARKS IN SINGLE TRACK OF HWMCC 2011

Benchmark	# Step	# Result	$Zigzag$		$Dual$		$IncUST$		Type
			Induct	Total	Induct	Total	Induct	Total	
pdtswvqis10x6p0.smv	82	FALSE	41.06	58.93	14.73	31.76	**4.22**	**20.92**	0
pdtswvqis8x8p0.smv	66	FALSE	22.79	35.69	9.65	20.18	**1.78**	**12.95**	0
pdtswvsam6x8p0.smv	48	FALSE	12.07	23.68	16.78	31.23	**4.63**	**18.75**	0
pdtswvibs8x8p1.smv	39	TRUE	3.64	6.84	1.27	4.50	**0.42**	**3.19**	0
pdtswvroz10x6p1.smv	67	TRUE	13.75	22.38	11.98	20.53	**4.82**	**13.24**	0
pdtswvroz8x8p1.smv	55	TRUE	5.54	10.12	6.73	12.50	**2.83**	**8.74**	0
pdtswvroz8x8p2.smv	73	TRUE	13.27	**28.64**	18.29	116.26	**5.65**	105.81	0
pdtswvsam4x8p4.smv	46	TRUE	12.72	**17.87**	11.19	24.66	**6.10**	18.57	0
pdtswvsam6x8p1.smv	44	TRUE	9.64	16.21	11.39	17.98	**6.91**	**13.88**	0
pdtswvsam6x8p2.smv	44	TRUE	8.98	15.81	11.14	17.82	**8.27**	**15.54**	0
pdtswvsam6x8p3.smv	55	TRUE	18.33	**30.23**	31.93	50.37	**13.49**	33.53	0
bc57sensorsp0.smv	105	FALSE	81.95	199.22	100.49	204.53	**36.72**	**126.36**	0
bc57sensorsp0neg.smv	105	FALSE	141.24	214.18	95.77	191.36	**33.63**	**124.86**	0
bc57sensorsp1.smv	105	FALSE	118.33	204.24	176.14	277.63	**100.63**	**198.57**	0
bc57sensorsp1neg.smv	105	FALSE	111.56	**215.59**	201.65	318.47	**103.87**	218.92	0
bc57sensorsp2.smv	105	FALSE	**89.48**	218.50	223.47	325.74	100.92	**201.82**	0
bc57sensorsp2neg.smv	105	FALSE	104.97	224.55	198.76	311.69	**94.74**	**213.20**	0
bc57sensorsp3.smv	105	FALSE	104.51	225.79	203.78	306.80	**96.81**	**191.13**	0
pdtswvtma6x4p2.smv	37	TRUE	17.92	19.41	5.13	6.99	**3.99**	**5.94**	1
pdtswvtma6x4p3.smv	44	TRUE	24.67	26.61	5.20	7.94	**4.51**	**7.23**	1
pdtswvtma6x6p1.smv	37	TRUE	4.38	6.30	0.81	3.37	**0.59**	**3.13**	1
pdtswvtma6x6p2.smv	37	TRUE	35.62	37.62	14.73	17.23	**13.05**	**15.42**	1
pdtswvtma6x6p3.smv	44	TRUE	45.81	48.66	12.38	15.71	**10.77**	**14.11**	1
pdtvsar8multip29.smv	4	TRUE	5.05	5.32	1.41	1.62	**1.22**	**1.45**	1
pj2013.smv	9	TRUE	5.45	11.49	2.24	8.31	**1.82**	**7.87**	1
pj2019.smv	9	TRUE	23.44	31.48	16.66	23.47	**11.89**	**19.76**	1
visprodcellp22.smv	48	TRUE	158.03	165.70	28.82	40.11	**27.17**	**38.85**	2
neclaftp3001.smv	13	FALSE	751.86	769.54	289.60	304.25	**233.80**	**253.06**	2
neclaftp3002.smv	15	FALSE	828.27	855.62	357.49	371.63	**313.69**	**326.80**	2
bobsynth06neg.smv	29	FALSE	53.33	114.53	**38.34**	**67.32**	66.74	94.69	3
bobsynth08neg.smv	28	FALSE	74.94	151.85	30.18	72.54	**27.06**	**69.90**	3
bobsynth11neg.smv	17	FALSE	6.49	27.32	**5.42**	**14.62**	11.64	20.88	3
bobsynth12neg.smv	15	FALSE	6.53	21.38	**5.17**	**12.39**	5.33	12.58	3
bobsynth13neg.smv	18	FALSE	14.03	37.51	**13.18**	**25.10**	13.25	25.35	3
mentorbm1and.smv	11	FALSE	14.72	33.34	**13.86**	**22.93**	15.59	24.63	3
mentorbm1p10.smv	16	FALSE	42.80	80.94	**34.39**	**51.63**	48.03	63.26	3
mentorbm1p11.smv	14	FALSE	29.45	56.52	30.66	48.52	**23.70**	**41.57**	3
mentorbm1p12.smv	11	FALSE	12.00	30.16	**9.18**	**18.05**	12.20	21.05	3
bobpci215.smv	11	FALSE	1.03	17.59	0.14	**7.82**	**0.12**	7.83	4
abp4ptimo.smv	21	FALSE	1.31	13.05	0.75	4.12	**0.42**	**3.87**	4
abp4ptimoneg.smv	21	FALSE	0.60	**4.18**	0.49	6.18	**0.38**	6.11	4
irstdme4.smv	52	FALSE	22.31	**444.38**	4.79	516.76	**2.41**	519.14	4
irstdme5.smv	52	FALSE	34.29	722.66	13.54	391.65	**4.03**	**355.41**	4

divide the 43 instances into 5 groups by the label "Typ" in TABLE I, which are also displayed in Fig. 2. We discuss the 5 groups in detail in the following.

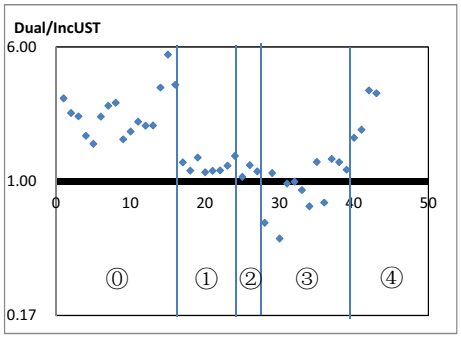
1) In Type 1, the properties are TRUE, and the main time of induction step is on the last UNSAT problem. So there are not much duplicate search in solving $Induct'_i(i \geq 0)$ sequently.
2) According to Algorithm 3, the search tree may backtrack when adding $Inc_{\mathbf{f}_{i-1},\mathbf{f}_i}$. In the instances of type 2, the algorithm backtracks to a high level, which clears most of the generated tree, so $IncUST$ gets only a little improvement on these instances.
3) When the algorithm backtracks to some high level, as it has learned some learnt clauses and changed the score of some variables, the variable decision order will be different from the original method, which makes the result unpredictable. Type 3 is in this situation, and for some instances of this type $Dual$ runs faster.
4) In Type 4, induction time accounts for a small portion of the total time, so it makes little sense to improve the induction time.
5) Type 0 does not have the problems of the above 4 cases. It is the case that's suitable for $IncUST$, and also the motivation of this paper.

VI. CONCLUSIONS

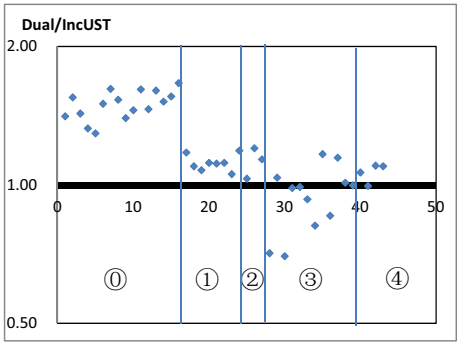
To reduce the duplicate workload in solving the problems of induction step in temporal induction, we propose an algorithm which can reuse the search tree of $Induct_k$ to help solving $Induct_{k+1}$. To realize this idea, the problems of induction step are converted to incremental CNFs by variable substitution. We further customize the DPLL procedure to support search tree reuse. Experimental results demonstrate the significant performance improvement of our algorithm in the induction step.

ACKNOWLEDGMENT

This work was supported by the Chinese National 973 Plan under grant No. 2010CB328003, the NSF of China under grants No. 61272001, 60903030, 91218302, the Chinese National Key Technology R&D Program under grant No. SQ2012BAJY4052, and the Tsinghua University Initiative Scientific Research Program.



(a) induction time



(b) total time

Fig. 2. Comparison between $IncUST$ and $Dual$

REFERENCES

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

[2] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi, "Benefits of bounded model checking at an industrial setting," in *Computer Aided Verification*. Springer, 2001, pp. 436–453.

[3] C. Van Eijk., "Sequential equivalence checking without state space traversal," in *Design, Automation and Test in Europe, 1998., Proceedings*. IEEE, 1998, pp. 618–623.

[4] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*. Springer, 2000, pp. 409–426.

[5] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*. Springer, 2000, pp. 127–144.

[6] N. Eén and N. Sorensson, "Temporal induction by incremental SAT solving," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003.

[7] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 542–545.

[8] H. Jin and F. Somenzi, "An incremental algorithm to check satisfiability for bounded model checking," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 119, no. 2, pp. 51–65, 2005.

[9] O. Shtrichman., "Tuning SAT checkers for bounded model checking," in *Computer Aided Verification*. Springer, 2000, pp. 480–494.

[10] C. Wang, H. Jin, G. Hachtel, and F. Somenzi, "Refining the SAT decision ordering for bounded model checking," in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 535–538.

[11] J. Hooker, "Solving the incremental satisfiability problem," *The Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.

[13] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *Journal of automated reasoning*, vol. 24, no. 1, pp. 67–100, 2000.

[14] M. Ben-Ari, A. Pnueli, and Z. Manna, "The temporal logic of branching time," *Acta informatica*, vol. 20, no. 3, pp. 207–226, 1983.

[15] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," *25 Years of Model Checking*, pp. 196–215, 2008.

[16] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic*, p. 115C125, 1962.

[17] D. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986.

[18] P. Jackson and D. Sheridan, "Clause form conversions for boolean circuits," in *Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 899–899.

[19] P. Manolios and D. Vroon, "Efficient circuit to CNF conversion," *Theory and Applications of Satisfiability Testing–SAT 2007*, pp. 4–9, 2007.

[20] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, 1960.

[21] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Comms. ACM*, vol. 5, pp. 394–397, July 1962.