Clause Replication and Reuse in Incremental Temporal Induction

Liangze Yin, Fei He, Ming Gu, and Jiaguang Sun
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, {FeiHe, guming}@mails.tsinghua.edu.cn

***Abstract*—Temporal induction is one of the most popular SAT-based model checking techniques. It consists of two parts, the base case and the induction step. With the search length increment, both parts generate a sequence of SAT problems. This paper focuses on learnt clause replication and reuse in incremental temporal induction. Firstly, with the aid of assumption literals, we present an alternative clause replication scheme, which is much easier to implement than existing works. Secondly, based on our clause replication scheme, we present several clause reuse schemes to maximally explore the learnt clauses and their replications in temporal induction. Based on above ideas, we propose two new incremental temporal induction algorithms. Experimental results on a large number of benchmarks show significant performance improvement of our technique.**

***Keywords*—Temporal induction; bounded model checking; Boolean satisfiability; clause learning;**

I. Introduction

Bounded model checking (BMC) [1] [2] is one of the most successful techniques to alleviate the state explosion problem. It tries to answer the question whether there is a counterexample within some given number of steps. This problem is then reduced to a problem of propositional satisfiability and solved by a propositional SAT solver. Temporal induction [3] [4] [5] [6] is an enhancement to BMC. Given a safety property, a proof of temporal induction [6] consists of two parts: the base case and the induction step. The base case acts as a "bug-finder", trying to disprove the property by finding a counterexample, and the induction step acts as a "prover", trying to prove the property by induction.

With the search length increment, both the base case and the induction step generate a sequence of SAT problems, called $Base$ sequence ($Base_k$, $k \geq 0$) and $Induct$ sequence ($Induct_k$, $k \geq 0$) respectively. During the SAT solving, conflict-driven clause learning has been proved an efficient technique. A learnt clause can prune the search space and then accelerate the search procedure. As the structures of the SAT problems generated in temporal induction are similar, a natural idea is to explore the structural similarities to maximally reuse the learnt clauses.

Observed that the transition relations of all time-frames in BMC are identical, Shtrichman proposed that, whenever a clause is learnt, some replicated clauses, which are new clauses symmetric to the original learnt clause, can be added to help prune later search space [7] [8]. However, the problems in BMC are asymmetric due to their initial and end formulas (the initial predicate $\mathcal{I}_0$ and the negation of the property $\neg\phi_k$). To solve this problem, he proposes to simulate an assignment for every potential replicated clause and checks if it leads to a conflict [7]. Alternatively, each clause is marked if it does not depend on $\mathcal{I}_0$ and $\neg\phi_k$, and a learnt clause can be replicated if all the clauses leading to the corresponding conflict are marked [8], which requires extra marking during the actual solving process. Some other works solve a sequence of problems with an incremental satisfiability engine [6] [9] [10]. The motivation is that the clauses learnt in one instance are not only useful in later search parts of the same SAT problem, but can also be used in later similar problems. To guarantee that all the learnt clauses can be safely added to subsequent problems, Een et al. proposed the scheme of setting assumption literals, which is very simple to implement and expressive enough to guarantee the correctness [6].

In this paper, with the aid of assumption literals proposed in [6], we present an alternate scheme of learnt clause replication. The idea is that, by setting the definition literals of all the initial, transition relation, and property predicates to be assumption literals, the learnt clauses are independent of any of the above parts, so all their replications can be safely added. This scheme is much easier to implement compared with the works in [7] [8]. There is no need for any simulation or extra marking. In other words, even if the initial and end formulas are considered, or the transition relations in different time-frames are different due to techniques like BCOI [11], we prove that all the replications of learnt clauses can still be safely added without any other computation.

Based on our clause replication scheme, we further propose that all the learnt clauses and their replications are not only useful in later search parts of the same problem, but can also be used to help solve subsequent problems. Actually, they can be reused in subsequent problems of the same sequence, and also in subsequent problems of the other sequence, even if the two sequences grow in different directions. In this sense, our work perfectly integrates the ideas of clause replication [7] [8] and incremental temporal induction [6].

Above techniques have been applied to two state-of-the-art temporal induction algorithms, $Dual$ and $Zigzag$ [6], which have been implemented in the NuSMV package. The new algorithms can make full use of clause replication and reuse. Experimental results on benchmarks from HWMCC'2011[1] show significant performance improvement of our technique.

[1] http://fmv.jku.at/hwmcc11/benchmarks.html

978-1-4799-5482-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ICECCS.2014.23

CPS Conference Publishing Services

The rest of this paper is organized as follows. In Section 2, we briefly review preliminaries on SAT solving and temporal induction. Then in Section 3 and Section 4, we present our clause replication and reuse schemes. Our incremental temporal induction algorithms are described in Section 5. Empirical evaluation is reported in Section 6. Finally, Section 7 concludes the paper.

II. Preliminaries

A. SAT Solving

Let $\mathbf{x}$ be a set of Boolean variables. A literal is either the positive- or negative-form of a variable, i.e., either x or $\neg x$, where $x \in \mathbf{x}$. A clause λ is a disjunction of literals. Denote ϵ the empty clause. Let Λ denote a set of clauses. We write $\Lambda \rightsquigarrow \lambda$ if λ can be deduced by Λ. A conjunctive normal form (CNF) is a conjunction of clauses. A CNF can be represented by the set of its involved clauses. Any propositional formula can be converted into a CNF in linear time by importing intermediate variables [12] [13] [14]. When a formula φ is converted into a CNF, the literal representing the whole formula is called its definition literal. Given a propositional formula φ and its definition literal p, we denote $[\varphi]^p$ the set of clauses defining φ (not including p), and $[\varphi]$ the set of clauses $[\varphi]^p \cup \{p\}$.

Most of modern SAT solvers are based on the DPLL algorithm [15] [16]. It is based on the exploration of the variable space, and can be seen as a Depth-First-Search (DFS) process. At each node, it selects a variable and assigns it true or false. This process is called making a *decision*. If a clause has only one free literal and all the other literals are false, then it is called a unit clause. The free literal of every 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). In the DPLL algorithm, it iteratively makes decisions and applies BCP, until either a satisfiable assignment is found or a conflict is encountered. If it is the latter case, the procedure backtracks and makes a new decision unless the search tree has been fully explored. For a DPLL algorithm, the number of BCPs reflects the size of the search tree and is usually used to evaluate the efficiency of a SAT solver.

In last decades, clause learning has been substantiated as one of the most encouraging techniques in modern SAT solvers. It is crucial to enable SAT solvers to solve large instances encoding real-world problems. The idea is that whenever a conflict is detected during the search process, it analyzes the reason for the conflict and encodes it as a so called learnt clause. All the learnt clauses are managed in a learnt clause database, which acts as a cache to prune the search space in later search parts.

B. Temporal Induction

Let a Finite State Machine (FSM) be $M(\mathbf{x}, \mathcal{I}(\mathbf{x}), \mathcal{R}(\mathbf{x}, \mathbf{x}'))$, where $\mathbf{x}$ and $\mathbf{x}'$ denote respectively the sets of current and next-state variables, $\mathcal{I}(\mathbf{x})$ is the initial condition predicate, and $\mathcal{R}(\mathbf{x}, \mathbf{x}')$ is the transition relation predicate. Denote $\phi(\mathbf{x})$ the safety property to be checked on M. 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 time-frames. In detail, $\mathbf{x}_i$ $(i \geq 0)$ represents the set of state variables at the i-th time-frame, and $x_i^j \in \mathbf{x}_i$ is the corresponding variable of x^j at the i-th time-frame. ϕ_i, $\mathcal{I}_i$ and $\mathcal{R}_i$ are used as shorthands to denote $\phi(\mathbf{x}_i)$, $\mathcal{I}(\mathbf{x}_i)$ and $\mathcal{R}(\mathbf{x}_i, \mathbf{x}_{i+1})$ respectively.

A temporal induction proof [5] consists of two parts: the base case and the induction step. The proof for the base case on length k can be formulated as

$$Base_k = \mathcal{I}_0 \wedge (\bigwedge_{i=0}^{k-1} (\mathcal{R}_i \wedge \phi_i)) \wedge (\neg \phi_k). \quad (1)$$

The proof for the induction step on length k can be formulated as

$$Induct_k = (\bigwedge_{i=0}^{k} (\mathcal{R}_i \wedge \phi_i)) \wedge (\neg \phi_{k+1}) \wedge Unique_{k+1}, \quad (2)$$

where

$$Unique_k = (\bigwedge_{0 \leq i < j \leq k} \mathbf{x}_i \neq \mathbf{x}_j) = (\bigwedge_{0 \leq i < j \leq k} (\bigvee_{t=1}^{n} (x_i^t \neq x_j^t))).$$

In temporal induction, the trace of states can either start from the initial states and then go forward along the transition relation, or start from the bad states and then go backward along the transition relation. Especially, proving the induction step in backward direction may get good results in some cases [6]. Let $\mathcal{R}_i'$ be a short hand of $\mathcal{R}(\mathbf{x}_{i+1}, \mathbf{x}_i)$. The proof for the induction step in backward direction can be formulated as

$$InductBack_k = (\neg \phi_0) \wedge (\bigwedge_{i=0}^{k} (\mathcal{R}_i' \wedge \phi_{i+1})) \wedge Unique_{k+1}, \quad (3)$$

where $Unique_k$ is defined as same as in the forward version.

Model checking based on temporal induction is straightforward. It repeatedly checks the satisfiability of $Base_k$ and $Induct_k$. Here length k starts from 0 and 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).

In [6], Een et al. proposed a technique called assumption literals. It assumes some literals (namely unit clauses) to be true before starting the SAT solving. Given a problem φ and a set of literals A, if the SAT solver regards all the literals in A as internal assumptions when solving φ, then all the clauses learnt in the search procedure are independent of those literals in A. As a result, when solving another SAT problem φ' that contains all the clauses in φ except those unit clauses in A, all the clauses learnt in solving φ can be reused to help solve φ'. For example, suppose that z and p_k are the definition literals of $\mathcal{I}_0$ and $\neg \phi_k$ respectively, when solving $Base_k$ in (1), we give $[\mathcal{I}_0]^z$, $[\neg \phi_k]^{p_k}$, and $[\bigwedge_{i=0}^{k-1} (\mathcal{R}_i \wedge \phi_i)]$ to a SAT solver, and make $\{z, p_k\}$ as assumption literals. Note that $\bigwedge_{i=0}^{k-1} (\mathcal{R}_i \wedge \phi_i)$ is also a part of $Base_{k+1}$. All the learnt clauses in solving $Base_k$ can be safely added to help solve $Base_{k+1}$.

With this technique, Een et al. provided two interface functions for a SAT solver:

- $addClause(\Lambda)$ adds a set of clauses to the solver; and

- $solve(Assump)$ solves the current instance by assuming all literals included in the set $Assump$ to be true.

Then the solving for the SAT problems in temporal induction can be performed in an incremental fashion.

Two incremental temporal induction algorithms, $Zigzag$ and $Dual$, have been proposed in [6]. In the $Zigzag$ algorithm, the proofs for the base case and for the induction step are merged together, and only one SAT solver is involved to solve them. In the $Dual$ algorithm, two different SAT solvers are used to solve these two sequences of problems respectively. Additionally, in the $Dual$ algorithm, the proof for the base case starts from the initial states and go forward, while the proof for the induction step can start from the bad states and go backward.

III. Formula Replication

Definition 1 (Shift Formula): Given a constant m and a propositional formula $\varphi(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_l)$ where $\mathbf{x}_i$ is the set of state variables at the i-th time-frame, $\varphi_{\gg m}$ is the shift formula of φ obtained by substituting each variable x_i^j $(0 \le i \le l, 0 \le j < n)$ in φ with x_{i+m}^j.

For example, if $\varphi = \neg x_2 \vee y_3 \vee z_5$, then $\varphi_{\gg 2} = \neg x_4 \vee y_5 \vee z_7$. Note that the value of m can be negative as long as $i \ge 0$ for any variable x_i^j in $\varphi_{\gg m}$. In above example, $\varphi_{\gg -2} = \neg x_0 \vee y_1 \vee z_3$. Especially, $\varphi_{\gg 0}$ is φ, and $\epsilon_{\gg m}$ is ϵ for any m.

Lemma 1: If $[\varphi] \rightsquigarrow \lambda$, then $[\varphi_{\gg m}] \rightsquigarrow \lambda_{\gg m}$, where m is a constant.

Proof: If $[\varphi] \rightsquigarrow \lambda$, there must exist a proof leading from $[\varphi]$ to λ, denote it by Θ. A proof leading from $[\varphi_{\gg m}]$ to $\lambda_{\gg m}$ can be obtained by substituting each variable x_i^j in Θ to x_{i+m}^j. Then the conclusion holds. ∎

Definition 2 (Reverse Formula): Given a constant m and a propositional formula $\varphi(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_l)$ where $\mathbf{x}_i$ is the set of state variables at the i-th time-frame, $\varphi_{!m}$ is the reverse formula of φ obtained by substituting each variable x_i^j $(0 \le i \le l, 0 \le j < n)$ in φ with x_{m-i}^j.

For example, if $\varphi = \neg x_2 \vee y_3 \vee z_5$, then $\varphi_{!8} = \neg x_6 \vee y_5 \vee z_3$. Again, the value of m must satisfy that $i \ge 0$ for any variable x_i^j in $\varphi_{!m}$. Especially, $\epsilon_{!m}$ is ϵ for any m.

Lemma 2: If $[\varphi] \rightsquigarrow \lambda$, then $[\varphi_{!m}] \rightsquigarrow \lambda_{!m}$, where m is a constant.

Proof: If $[\varphi] \rightsquigarrow \lambda$, there must exist a proof leading from $[\varphi]$ to λ, denote it by Θ. A proof leading from $[\varphi_{!m}]$ to $\lambda_{!m}$ can be obtained by substituting each variable x_i^j in Θ to x_{m-i}^j. Then the conclusion holds. ∎

Given a propositional formula φ, any shift formula or reverse formula of φ is called a replication of φ.

IV. Clause Replication and Reuse

A. Clause Reuse for Base Sequence

For any clause learnt in solving $Base_k$, to guarantee that all of its shifted clauses can be safely added to help solve later parts of $Base_k$ and subsequent problems, we solve $Base_k$ as in Algorithm 1. Different from traditional approaches, here all definition literals of $\mathcal{I}_0$, $\mathcal{R}_i$ $(i < k)$, ϕ_i $(i < k)$ and $\neg\phi_k$, which are denoted by z, r_i $(i < k)$, p_i $(i < k)$ and q_k respectively, are made as assumption literals. In this manner, all the learnt clauses of $Base_k$ are independent of $\mathcal{I}_0$, $\mathcal{R}_i$ $(i < k)$, ϕ_i $(i < k)$ and $\neg\phi_k$.

Algorithm 1 $solveBase(Base_k)$

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ, step k
1: $addClause([\mathcal{I}_0]^z)$
2: $Assump \leftarrow \{z\}$
3: **for** $i = 0$ to $k-1$ **do**
4: $\quad addClause([\mathcal{R}_i]^{r_i}, [\phi_i]^{p_i})$
5: $\quad Assump \leftarrow Assump \cup \{r_i, p_i\}$
6: **end for**
7: $addClause([\neg\phi_k]^{q_k})$
8: $Assump \leftarrow Assump \cup \{q_k\}$
9: **return** $solve(Assump)$

Theorem 1: In Algorithm 1, for any learnt clause λ of $Base_k$, the satisfiability of $[Base_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[Base_{k+n}]$ where $n \ge 0$.

Proof: To prove that the satisfiability of $[Base_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[Base_{k+n}]$, we try to find a formula Δ such that $\Delta \rightsquigarrow \lambda_{\gg m}$ and the satisfiability of $[Base_{k+n}] \cup \Delta$ is equivalent with that of $[Base_{k+n}]$.

Let Ψ be the set of clauses according to definition literals of $\mathcal{I}_0$, $\mathcal{R}_i$ $(i < k)$, ϕ_i $(i < k)$ and $\neg\phi_k$, and $\Gamma = [Base_k] \setminus \Psi$. As λ is a learnt clause of $Base_k$, and all the literals in Ψ are made as assumption ones, $\Gamma \rightsquigarrow \lambda$. According to Lemma 1, $\Gamma_{\gg m} \rightsquigarrow \lambda_{\gg m}$. The following proves that the satisfiability of $[Base_{k+n}] \cup \Gamma_{\gg m}$ is equivalent with that of $[Base_{k+n}]$.

$\Gamma_{\gg m} = \{[\mathcal{I}_0]^z, \bigcup_{0 \le i < k}[\mathcal{R}_i]^{r_i}, \bigcup_{0 \le i < k}[\phi_i]^{p_i}, [\neg\phi_k]^{q_k}\}_{\gg m}$. As $[\mathcal{I}_0]^z_{\gg m}$ means that $z_{\gg m} \Leftrightarrow (\mathcal{I}_0)_{\gg m}$, adding $[\mathcal{I}_0]^z_{\gg m}$ to $[Base_{k+n}]$ does not affect the satisfiability of $[Base_{k+n}]$. Similarly, we can prove that adding $\Gamma_{\gg m}$ to $[Base_{k+n}]$ does not affect the satisfiability of $[Base_{k+n}]$.

Thus we find a formula $\Delta = \Gamma_{\gg m}$ such that $\Delta \rightsquigarrow \lambda_{\gg m}$ and the satisfiability of $[Base_{k+n}] \cup \Delta$ is equivalent with that of $[Base_{k+n}]$. Then the conclusion holds. ∎

In Theorem 1, if $m = 0$ and $n = 0$, it means that learnt clauses can be used to help solve the current problem, which is the idea of conflict clause learning; if $m = 0$ and $n > 0$, it means that learnt clauses can also be used to help solve subsequent problems, just as discussed in traditional incremental temporal induction [6]; if $m \neq 0$ and $n = 0$, it means that the shifted ones of learnt clauses can be added to help solve the current problem, which is the idea of clause replication [7]; and if $m \neq 0$ and $n > 0$, it means that the shifted ones of learnt clauses can also be used to help solve subsequent problems. From Theorem 1, both the ideas of clause replication and incremental temporal induction are perfectly integrated together in our method.

Another thing we need to note here is that the time-frame scope in $Base_{k+n}$ is $[0, k+n]$. For any clause $\lambda_{\gg m}$, if its subscripts exceed the time-frame scope, $\lambda_{\gg m}$ will never be a conflict clause, and it is unwanted in the solving process of $Base_{k+n}$. So, when we add the replication $\lambda_{\gg m}$ to help solve

$Base_{k+n}$, the value of m must satisfy that the subscribes of $\lambda_{\gg m}$ cannot exceed the time-frame scope of $Base_{k+n}$. We call this *the time-frame constraint*.

From Algorithm 1, our approach is very easy to implement. Though the SAT problem is not fully symmetrical due to $\mathcal{I}_0$ and $\neg\phi_k$, we proved in Theorem 1 that all learnt clauses and their shifted ones can be safely added to help solve the current and subsequent problems without any other computation. Compared with the works in [7] [8], our method has another two advantages. First, more learnt clauses are considered in our method. For example, in his works, any leant clause λ related with $\neg\phi_k$ is not considered. Actually, for each subsequent problem, there is one shifted clause of λ which is useful to prune the search space. Second, in his work, a hypothesis is that the transition relations of different time-frames should be identical; while our method still works even if this is not the truth due to improvement techniques like BCOI [11].

B. Clause Reuse for Induction Sequence

Similar to base sequence, when solving $Induct_k$, we make all the definition literals of $\mathcal{R}_i$ $(i < k)$, ϕ_i $(i \leq k)$, and $\neg\phi_{k+1}$ as assumption literals. The approach is shown in Algorithm 2. The algorithm for solving $InductBack_k$ is similar. We make all the definition literals of $\neg\phi_0$, $\mathcal{R}'_i$ $(i < k)$, and ϕ_{i+1} $(i < k)$ as assumption literals. The approach is shown in Algorithm 3.

Algorithm 2 $solveInduct(Induct_k)$

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ, step k
1: $Assump \leftarrow \emptyset$
2: **for** $i = 0$ to $k-1$ **do**
3: $\quad addClause([\mathcal{R}_i]^{r_i}, [\phi_i]^{p_i})$
4: $\quad Assump \leftarrow Assump \cup \{r_i, p_i\}$
5: **end for**
6: $addClause([\neg\phi_{k+1}]^{q_{k+1}})$
7: $Assump \leftarrow Assump \cup \{q_{k+1}\}$
8: $addClause([Unique_{k+1}])$
9: **return** $solve(Assump)$

Algorithm 3 $solveInductBack(InductBack_k)$

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ, step k
1: $addClause([\neg\phi_0]^{q_0})$
2: $Assump \leftarrow \{q_0\}$
3: **for** $i = 0$ to $k-1$ **do**
4: $\quad addClause([\mathcal{R}'_i]^{r'_i}, [\phi_{i+1}]^{p_{i+1}})$
5: $\quad Assump \leftarrow Assump \cup \{r'_i, p_{i+1}\}$
6: **end for**
7: $addClause([Unique_{k+1}])$
8: **return** $solve(Assump)$

Similar as for Algorithm 1, we can prove the following theorems for Algorithm 2 and Algorithm 3 respectively.

Theorem 2: In Algorithm 2, for any learnt clause λ of $Induct_k$, the satisfiability of $[Induct_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[Induct_{k+n}]$ where $n \geq 0$.

Theorem 3: In Algorithm 3, for any learnt clause λ of $InductBack_k$, the satisfiability of $[InductBack_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[InductBack_{k+n}]$ where $n \geq 0$.

Similar to base sequence, when a shifted clause $\lambda_{\gg m}$ is added to subsequent problems, it must satisfy the time-frame constraint.

C. Clause Sharing between Base and Induction Sequence

In previous subsections, we discussed the clause replication and reuse in one sequence. Observed that the problems in $Base$ and $Induct$ sequences are also similar, this section considers the clause sharing between the two sequences. Denote $Base$, $Induct$ and $InductBack$ the sets of SAT problems $\{Base_k | 0 \leq k \leq \infty\}$, $\{Induct_k | 0 \leq k \leq \infty\}$, and $\{InductBack_k | 0 \leq k \leq \infty\}$ respectively. This section discusses the clause reuse in two situations: clause reuse between $Base$ and $Induct$, and clause reuse between $Base$ and $InductBack$.

Theorem 4: In Algorithm 1, for any learnt clause λ of $Base_k$, the satisfiability of $[Induct_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[Induct_{k+n}]$ where $n \geq 0$. In Algorithm 2, for any learnt clause λ of $Induct_k$, the satisfiability of $[Base_{k+n}] \cup \{\lambda_{\gg m}\}$ is equivalent with that of $[Base_{k+n}]$ where $n \geq 0$.

Proof: The proof is similar to that of Theorem 1. We can construct a formula $\Delta = \Gamma_{\gg m}$ such that $\Delta \rightsquigarrow \lambda_{\gg m}$ and the satisfiability of $[Induct_{k+n}] \cup \Delta$ is equivalent with that of $[Induct_{k+n}]$. The reverse is similar. ■

Theorem 5: In Algorithm 1, for any learnt clause λ of $Base_k$, the satisfiability of $[InductBack_{k+n}] \cup \{\lambda_{!m}\}$ is equivalent with that of $[InductBack_{k+n}]$ where $n \geq 0$. In Algorithm 3, for any learnt clause λ of $InductBack_k$, the satisfiability of $[Base_{k+n}] \cup \{\lambda_{!m}\}$ is equivalent with that of $[Base_{k+n}]$ where $n \geq 0$.

Proof: The proof is similar to that of Theorem 1. We can construct a formula $\Delta = \Gamma_{!m}$ such that $\Delta \rightsquigarrow \lambda_{!m}$ and the satisfiability of $[InductBack_{k+n}] \cup \Delta$ is equivalent with that of $[InductBack_{k+n}]$. The reverse is similar. ■

V. Incremental Temporal Induction

Thanks to the theorems proved in Section IV, model checking by temporal induction can be organized in an incremental fashion. Two incremental temporal induction algorithms are proposed: one proves the base case and the induction step with one SAT solver, called $Zigzag^+$; the other proves these two parts with different SAT solvers, called $Dual^+$.

Based on Theorem 1, 2 and 3, we revise the interface function $solve()$ of a SAT solver to $solve^+()$. Whenever a learnt clause λ is generated, both λ and its replications satisfying the time-frame constraint are added to the learnt clause database. Moreover, as the current problem widens the time-frame scope, for each learnt clause of previous problems, a new replication satisfying current time-frame constraint can be generated. This process is finished in function $expand()$. At the beginning of $solve^+()$, $expand()$ is used to generate the new replications and adds them to the learnt clause database.

$Dual^+$ consists of two parts: the algorithm for the base case and that for the induction step. According to the directions of these two parts, there can be up to 4 versions of $Dual^+$. Here we let the base case grow in forward direction (Algorithm 4) and the induction step grow in backward direction

(Algorithm 5). Note that in these algorithms, after one run of SAT solving (Line 7 in Algorithm 4, Line 6 in Algorithm 5), all learnt clauses and their replications are safely kept and passed to the next SAT solving.

Note that at Line 5 in Algorithm 4, Δ_{k-1} represents the set of learnt clauses of $InductBack_{k-1}$, and $(\Delta_{k-1})_{!k} = \{\delta_{!k} | \delta \in \Delta_{k-1}\}$ denotes the reversed versions of Δ_{k-1}. Λ_k in Algorithm 5 is similar (Line 4). It represents the set of learnt clauses of $Base_k$, and $(\Lambda_k)_{!k} = \{\lambda_{!k} | \lambda \in \Lambda_k\}$.

Compared with *Dual* proposed in [6], *Dual*$^+$ can learn much more clauses. First, whenever a learnt clause is generated, all its replications satisfying the time-frame constraint are also considered. Second, for all the learnt clauses of previous problems, their replications satisfying current time-frame constraint are also considered. Third, the learnt clauses of the other sequence and their replication are also shared, even if the two sequences grow in different directions.

Algorithm 4 *Dual*$^+$: base case in forward direction

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ
1: $addClause([\mathcal{I}_0]^z)$
2: $Assump \leftarrow \{z\}$
3: **for** $k = 0$ to ∞ **do**
4: $\quad addClause([\phi_k]^{p_k})$
5: $\quad addClause((\Delta_{k-1})_{!k})$
6: $\quad expand()$
7: $\quad$ **if** $solve^+(Assump \cup \{\neg p_k\}) = SAT$ **then**
8: $\quad\quad$ **return** $false$
9: $\quad$ **end if**
10: $\quad addClause([\mathcal{R}_k]^{r_k}])$
11: $\quad Assump \leftarrow Assump \cup \{r_k, p_k\}$
12: **end for**

Algorithm 5 *Dual*$^+$: induction step in backward direction

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ
1: $addClause([\phi_0]^{p_0})$
2: $Assump \leftarrow \{\neg p_0\}$
3: **for** $k = 0$ to ∞ **do**
4: $\quad addClause((\Lambda_k)_{!k})$
5: $\quad expand()$
6: $\quad$ **if** $solve^+(Assump) = UNSAT$ **then**
7: $\quad\quad$ **return** $true$
8: $\quad$ **end if**
9: $\quad addClause([\mathcal{R}'_k]^{r'_k}, [\phi_{k+1}]^{p_{k+1}})$
10: $\quad Assump \leftarrow Assump \cup \{r'_k, p_{k+1}\}$
11: $\quad$ **for** $i = 0$ to k **do**
12: $\quad\quad addClause([\mathbf{x}_i \neq \mathbf{x}_{k+1}])$
13: $\quad$ **end for**
14: **end for**

Zigzag$^+$ is shown in Algorithm 6. Note that in both *Zigzag* and *Zigzag*$^+$, the search directions of the base case and the induction step must be same. Here both the base case and the induction step are in forward direction. As these two parts are solved in the same solver instance, their clauses are naturally shared. Similar to *Dual*$^+$, *Zigzag*$^+$ learns much more clauses than *Zigzag*.

Algorithm 6 *Zigzag*$^+$: both base case and induction step in forward direction

Require: model $M(\mathbf{x}, \mathcal{I}, \mathcal{R})$, property ϕ
1: $addClause([\mathcal{I}_0]^z)$
2: **for** $k = 0$ to ∞ **do**
3: $\quad addClause([\phi_k]^{p_k})$
4: $\quad$ // induction step
5: $\quad expand()$
6: $\quad$ **if** $solve^+(Assump \cup \{\neg p_k\}) = UNSAT$ **then**
7: $\quad\quad$ **return** $true$
8: $\quad$ **end if**
9:
10: $\quad$ // base case
11: $\quad$ **if** $solve^+(Assump \cup \{z, \neg p_k\}) = SAT$ **then**
12: $\quad\quad$ **return** $false$
13: $\quad$ **end if**
14: $\quad addClause([\mathcal{R}_k]^{r_k})$
15: $\quad Assump \leftarrow Assump \cup \{r_k, p_k\}$
16: $\quad$ **for** $i = 0$ to k **do**
17: $\quad\quad addClause([\mathbf{x}_i \neq \mathbf{x}_{k+1}])$
18: $\quad$ **end for**
19: **end for**

VI. Empirical Evaluation

We implement *Dual*$^+$ and *Zigzag*$^+$ in NuSMV (version 2.5.3)[2], which already contains *Dual* and *Zigzag*. We compare the performance of our algorithms with those of them. The involved SAT solver is minisat2 (version 070721)[3]. All experiments are conducted on a 2.67GHz Intel Core (TM) CPU with 4GB memory.

All 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[4]. The single track has 467 instances in total, where 400 instances are publicly available (the other 67 instances from Intel have license restrictions). Moreover, we ignore those instances that can be solved in less than 5 seconds by all algorithms (including 134 instances) and those ones that cannot be solved within 300 seconds by any algorithm (including 232 instances). The remaining 34 instances are taken into consideration.

The experimental results are listed in Table I, where `No` column gives the instance No., `Benchmark` column shows the name of the instance, `Step` column lists the steps for the model checker to find a counterexample or to prove the property, `Result` column presents the validation of the property, `Zigzag`, `Zigzag`$^+$, `Dual`, and `Dual`$^+$ columns list the run time (in seconds) of these four algorithms respectively, `Z-Ratio` column gives the quotient of `Zigzag` to `Zigzag`$^+$, and `D-Ratio` column gives the quotient of `Dual` to `Dual`$^+$. The bold fonts indicate the best results among the four algorithms. For each algorithm, "-" indicates the model checking runs out of memory or exceeds the time limit.

From the table, our strategy significantly speeds up both *Zigzag* and *Dual*. Among the 34 instances, *Zigzag*$^+$ wins over *Zigzag* for 33 instances, and it runs more than 1.5 times faster

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

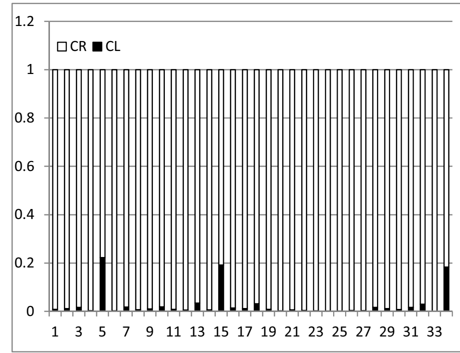
TABLE I. EXPERIMENTAL RESULTS FOR BENCHMARKS IN SINGLE TRACK OF HWMCC 2011

No	Benchmark	Step	Result	$Zigzag$	$Zigzag^+$	Z-Ratio	$Dual$	$Dual^+$	D-Ratio
1	mentorbm1p12.smv	12	false	13.39	**8.41**	1.6	8.66	12.66	0.7
2	mentorbm1and.smv	12	false	19.69	**12.38**	1.6	14.62	15.44	0.9
3	bobsynth13neg.smv	19	false	16.6	15.56	1.1	18.49	**11.36**	1.6
4	pdtswvqis8x8p0.smv	67	false	22.85	**10.26**	2.2	26.15	11.74	2.2
5	abp4ptimo.smv	22	false	5.32	4.85	1.1	**2.26**	3.38	0.7
6	pdtswvqis10x6p0.smv	83	false	35.83	**12.98**	2.8	47.37	19.8	2.4
7	bobsynth07neg.smv	25	false	31.51	33.82	0.9	33.74	**29.33**	1.2
8	mentorbm1p10.smv	17	false	43.8	33.54	1.3	41.49	**29.26**	1.4
9	pdtswvsam6x8p0.smv	49	false	18.91	**9.2**	2.1	28.31	17.29	1.6
10	bobsynth11neg.smv	18	false	12.06	**10.79**	1.1	12.48	13.96	0.9
11	bobsynth06neg.smv	30	false	99.19	46.98	2.1	64.16	**41.34**	1.6
12	bobsynthetic2.smv	13	false	136.51	72.75	1.9	156.84	**33.21**	4.7
13	bobsynth08neg.smv	29	false	101.11	53.93	1.9	88.52	**51.7**	1.7
14	mentorbm1p11.smv	15	false	42.17	**24.85**	1.7	26.67	31.07	0.9
15	bobtuttt.smv	29	false	45.8	**28.1**	1.6	66.7	57.36	1.2
16	pdtswvroz10x6p1.smv	68	true	14.37	11.25	1.3	18.31	**11.09**	1.7
17	pdtswvroz8x8p1.smv	56	true	7.03	**6.39**	1.1	10.27	6.61	1.6
18	pdtswvroz8x8p2.smv	74	true	21.13	**19.06**	1.1	127.74	31.67	4
19	pdtswvsam4x8p4.smv	47	true	16.13	10.88	1.5	24.7	**10.74**	2.3
20	pdtswvsam6x8p1.smv	45	true	13.01	**7.39**	1.8	15.35	9.18	1.7
21	pdtswvsam6x8p2.smv	45	true	12.99	**8.11**	1.6	13.81	10.27	1.3
22	pdtswvsam6x8p3.smv	56	true	28.16	**14.3**	2	42.09	16.65	2.5
23	pdtswvtma6x4p2.smv	38	true	12.2	**6.97**	1.8	9.72	7.53	1.3
24	pdtswvtma6x4p3.smv	45	true	14.02	**7.64**	1.8	18.77	8.41	2.2
25	pdtswvtma6x6p1.smv	38	true	4.21	**1.73**	2.4	5.27	1.88	2.8
26	pdtswvtma6x6p2.smv	38	true	24.72	**23.32**	1.1	39.35	26.01	1.5
27	pdtswvtma6x6p3.smv	45	true	23.81	22.97	1	43.5	**19.33**	2.3
28	visprodcellp22.smv	49	true	96.76	20.39	4.7	119.5	**14.99**	8
29	pdtswvqis10x6p1.smv	154	true	-	210.6	> 1	-	246.71	> 1
30	pdtswvqis8x8p1.smv	108	true	175.71	**70.84**	2.5	-	88.97	> 1
31	pdtswvsam6x8p4.smv	67	true	57.55	**34.26**	1.7	125.84	38.34	3.3
32	pdtswvroz10x6p2.smv	90	true	58.28	**48.61**	1.2	-	73.09	> 1
33	pj2013.smv	10	true	8.61	6.13	1.4	5.84	**5.51**	1.1
34	pj2019.smv	10	true	24.6	**14.86**	1.7	28.92	25.94	1.1

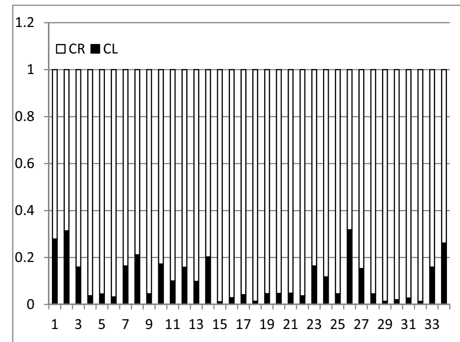
than $Zigzag$ for 21 instances. For $Dual^+$ and $Dual$, $Dual^+$ beats $Dual$ for 29 instances, and it runs more than 1.5 times faster than $Dual$ for 20 instances. For the other 5 instances, $Dual^+$ runs just a bit slower than $Dual$. Also note that there are 3 instances that $Dual$ cannot verify within the time limit.

To further analyse the performance of our strategy, more information about solving the 34 instances by $Dual^+$ is presented. Let D be the total amount of learnt clauses. Denote CL and CR the sets of learnt clauses generated by original conflict-driven learning and clause replication respectively. Obviously, $D = CL \cup CR$. We plot in Fig. 1 the percentages of CL and CR in D when the instance is solved, 1(a) for the base case and 1(b) for the induction step. In these two figures, x-axis represents the instance ID, and y-axis indicates the percentages. The black bars stand for the parts of CL, and the white bars stand for the parts of CR. From the figure, we can see that $Dual^+$ generates a mass of clauses with our clause replication and reuse technique. For most instances in the base case, more than 98% of the learnt clauses are originated from the clause replication.

We say a learnt clause is "valid" if it is a reason of some later conflict. Here the conflict may occur in the search process of the current or subsequent problems. Fig. 2 demonstrates the number of "valid" clauses in D, 2(a) for the base case and 2(b) for the induction step. In the two figures, x-axis represents the instance ID, and y-axis indicates the percentages of "valid" clauses. The black bars stand for the "valid" clauses from CL, and the white bars stand for the "valid" clauses from CR. From the figure, we observe that: (1) most of "valid" clauses come from CR; (2) the total amount of "valid" clauses hold a small portion of D. According to these observations, the good news is that the replications of learnt clauses really play

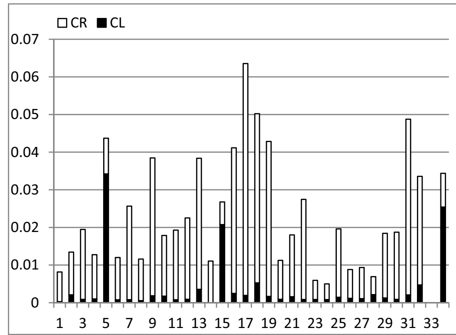


(a) base case

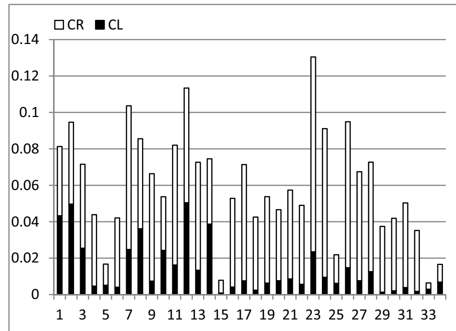


(b) induct step

Fig. 1. Size of CL vs. size of CR

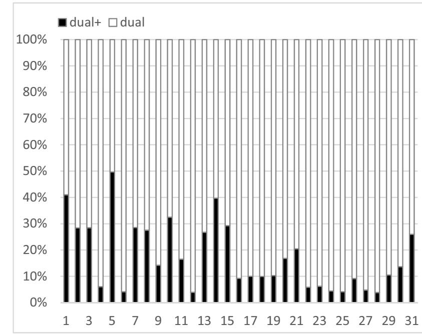


(a) base case

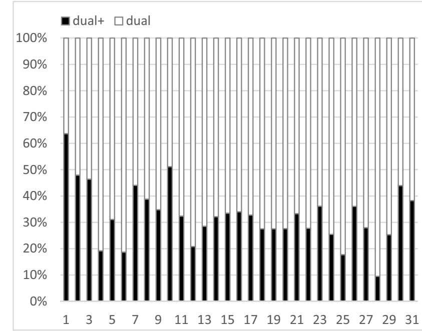


(b) induct step

Fig. 2. The percentage of "valid" clauses



(a) base case



(b) induct step

Fig. 3. Numbers of BCPs for $Dual$ and $Dual^+$

a positive role during the search process, and the bad news is that many clauses are never used in current or subsequent problems, which may instead slow down the process of finding unit clauses. Actually, for many instances, up to 99% of the clauses in D are never used.

Fig. 3 compares $Dual^+$ with $Dual$ in the number of BCPs (Boolean Constraint Propagation). For each instance, 1 in y-axis represents the total amount of BCPs of $Dual^+$ and $Dual$. The black and white bars represent the parts of BCPs for $Dual^+$ and $Dual$, respectively. From the figure, the number of BCPs of $Dual^+$ is much less than that of $Dual$ in most cases. This indicates that though most of the clauses in CR are never used, benefit from the minority "valid" clauses shown in Fig. 2, the propagations can still be greatly reduced.

From above analysis, our technique generates a lot of clauses, and many of them have been selected as "valid" ones. However, the amount of "valid" clauses account for only a small portion. Though most SAT solvers have integrated the "bad" clause identification and deletion technique. These strategies are not good enough, and this is still an interesting and challenging problem in our technique.

VII. Conclusions

This paper considers safety property verification by temporal induction. We present a new clause replication and reuse technique. It is much easier to implement and more powerful than existing works. With this technique, we realized two new incremental temporal induction algorithms. Experimental results on a large number of benchmarks show significant performance improvement of our algorithms.

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.

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*, pages 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*, pages 436–453. Springer, 2001.

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

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

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

[6] N. Eén and N. Sorensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[7] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification*, pages 480–494. Springer, 2000.

[8] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. *Correct Hardware Design and Verification Methods*, pages 58–70, 2001.

[9] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference, 2001. Proceedings*, pages 542–545. IEEE, 2001.

[10] H.S. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 119(2):51–65, 2005.

[11] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a powerpc- microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification*, pages 60–71. Springer, 1999.

[12] D.A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[13] P. Jackson and D. Sheridan. Clause form conversions for boolean circuits. In *Theory and Applications of Satisfiability Testing*, pages 899–899. Springer, 2005.

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

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

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