Efficient SAT Solving under Assumptions

Alexander Nadel[1] and Vadim Ryvchin[1,2]

[1] Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
{alexander.nadel,vadim.ryvchin}@intel.com
[2] Information Systems Engineering, IE, Technion, Haifa, Israel

Abstract. In incremental SAT solving, assumptions are propositions that hold solely for one specific invocation of the solver. Effective propagation of assumptions is vital for ensuring SAT solving efficiency in a variety of applications. We propose algorithms to handle assumptions. In our approach, assumptions are modeled as unit clauses, in contrast to the current state-of-the-art approach that models assumptions as first decision variables. We show that a notable advantage of our approach is that it can make preprocessing algorithms much more effective. However, our initial scheme renders assumption-dependent (or temporary) conflict clauses unusable in subsequent invocations. To resolve the resulting problem of reduced learning power, we introduce an algorithm that transforms such temporary clauses into assumption-independent pervasive clauses. In addition, we show that our approach can be enhanced further when a limited form of look-ahead information is available. We demonstrate that our approach results in a considerable performance boost of the SAT solver on instances generated by a prominent industrial application in hardware validation.

1 Introduction

A variety of SAT applications require the ability to solve incrementally generated SAT instances online [1–7]. In such settings the solver is expected to be invoked multiple times. Each time it is asked to check the satisfiability status of all the available clauses under *assumptions* that hold solely for one specific invocation. The naïve algorithm which solves the instances independently is inefficient, since all learning is lost [1–4].

The current state-of-the-art approach to this problem was proposed in [4] and implemented in the MiniSat SAT solver [8]. MiniSat reuses a single SAT solver instance for all the invocations. Each time after solving is completed, the user can add new clauses to the solver and reinvoke it. The user is also allowed to provide the solver a set of *assumption literals*, that is, literals that are always picked as the first decision literals by the solver. In this scheme, all the conflict clauses generated are *pervasive*, that is, assumption-independent. We call this approach to the problem of incremental SAT solving under assumptions the *Literal-based Single instance (LS)* approach, since it reuses a single SAT solver instance and models assumptions as decision literals. The approach of [1] to our problem would use a separate SAT solver instance for each invocation, where

each assumption would be encoded as a unit clause. To increase the efficiency of learning, it would store and reuse the set of assumption-independent *pervasive* conflict clauses throughout all the SAT invocations. We call this approach the *Clause-based Multiple instances (CM)* approach, since it uses multiple SAT solver instances and models assumptions as unit clauses.

It was shown in [4] that LS outperforms CM in the context of model checking. As a result, LS is currently widely applied in practice (e.g. [5–7]). The goal of this paper is to demonstrate its limitations and to propose an efficient alternative.

This study springs from the authors' experiences, described herein, in tuning Intel's formal verification flow. Verification engineers reported to us that a critical property could not be solved by the SAT solver within two days. Our default flow used the LS approach, where to check a property the property's negation is provided as an assumption. The property holds iff the instance is unsatisfiable. Surprisingly, we discovered that providing the negation of the property as a unit clause, rather than as an assumption, rendered the property solvable within 30 minutes. The reason for this was that the unit clause triggered a huge simplification chain for our SatELite [9]-like preprocessor that drastically reduced the number of clauses in the formula.

Our experience highlights a drawback of LS: preprocessing techniques cannot propagate assumptions in LS, since they are modeled as decision variables, while assumptions can be propagated in CM, where they are modeled as unit clauses. Section 3 of this work demonstrates how to incorporate the SatELite algorithm within CM and shows why the applicability of SatELite for LS is an open problem.

LS has important advantages over CM related to the efficiency of learning. First, in LS all the conflict clauses are pervasive and can be reused, while CM cannot reuse *temporary* conflict clauses, that is, clauses that depend on assumptions. Second, LS reuses all the information relevant to guiding the SAT solver's heuristics, while CM has to gather relevant information from scratch for each new incremental invocation of the solver. Section 4 of this paper proposes an algorithm that overcomes the first of the above-mentioned drawbacks of CM: our algorithm transforms temporary clauses into pervasive clauses as a post-processing step. Section 5 introduces an algorithm that mitigates the second of the above-mentioned advantages of plain LS over CM, given that limited look-ahead information is available to the solver. In fact, we propose an algorithm that combines LS and CM to achieve the most efficient results.

We study the performance of algorithms for incremental SAT solving under assumptions on instances generated by a prominent industrial application in hardware validation, detailed in Section 2. Section 2 also provides some defintions and background information. Experimental results demonstrating the efficiency of our algorithms are provided in Section 6. We would like to emphasize that all the SAT instances used in this paper are publicly available from the authors. Section 7 concludes our work.

2 Background

An incremental SAT solver is provided with the input $\{F_i, A_i\}$ at each invocation i, where for each i, F_i is a formula in Conjunctive Normal Form (CNF) and $A_i = \{l_1, l_2, \ldots, l_n\}$ is a set (conjunction) of *assumptions*, where each assumption l_j is a unit clause (it is also a literal). Invocation i of the solver decides the satisfiability of $(\bigwedge_{j=1}^{i} F_j) \wedge A_i$. Intuitively, before each invocation the solver is provided with a new block of clauses and a set of assumptions. It is asked to solve a problem comprising all the clauses it has been provided with up to that moment under the set of assumptions relevant only to a single invocation of the solver. Modern SAT solvers generate *conflict clauses* by resolution over input clauses and previously generated conflict clauses. A clause α is *pervasive* if $(\bigwedge_{j=1}^{i} F_j) \rightarrow \alpha$, otherwise it is *temporary*.

The Clause-based Multiple instances (CM) approach [1] to incremental SAT solving under assumptions operates as follows. CM creates a new instance of a SAT solver for each invocation. Each invocation decides the satisfiability of $(\bigwedge_{j=1}^{i} F_j) \wedge (\bigwedge_{l=1}^{i-1} P_l) \wedge A_i$, where P_l is the set of pervasive conflict clauses generated at invocation l of the solver. To keep track of temporary and pervasive conflict clauses, the algorithm marks all the assumptions as temporary clauses and marks a newly generated conflict clause as temporary iff one or more temporary clauses participated in its resolution derivation.

The Literal-based Single instance (LS) approach [4] to incremental SAT solving under assumptions reuses the same SAT instance for all the invocations. The instance is always updated with a new block of clauses. The key idea is in providing the assumptions as *assumption literals*, that is, literals that are always picked as the first decision literals by the solver. Conflict-clause learning algorithms ensure that any conflict clause that depends on a set of assumptions will contain the negation of these assumptions. Hence, in LS all the conflict clauses are pervasive.

While all the algorithms for incremental SAT solving under assumptions discussed in this paper are application-independent, the experimental results section studies the performance of various algorithms on instances generated by the following prominent industrial application in hardware validation.

Assume that a verification engineer needs to formally verify a set of properties in some circuit up to a certain bound. Formal verification techniques cannot scale to large modern circuits, hence the engineer needs to select a sub-circuit and mimic the environment of the larger circuit by imposing assumptions (also called constraints) [10]. The engineer then invokes SAT-based Bounded Model Checking (BMC) [11] to verify a property under the assumptions. If the result is satisfiable, then either the environment is not set correctly, that is, assumptions are incorrect or missing, or there is a real bug. In practice the first reason is much more common than the second. To discover which of the possibilities is the correct one, the engineer needs to analyze the counter-example. If the reason for satisfiability lies in incorrect modeling of the environment, the assumptions must be modified and BMC invoked again. When one property has been verified,

the engineer can move on to another. Practice shows that most of the validation time is spent in this process, which is known as the *debug loop*.

In the standard industrial BMC-based formal validation flow the model checker instance is built from scratch for each iteration of the debug loop. The key idea behind our solution is to take advantage of incremental SAT solving under assumptions *across multiple invocations of the model checker*. We keep only one instance of the model checker. For each invocation of BMC, given a transition system Ψ, a safety property Δ, and a set of assumptions Λ, we check whether Ψ satisfies Δ given Λ at each bound up to a given bound k using incremental SAT solving under assumptions, as follows. At each bound i, the transition system Ψ unrolled to i is translated to CNF and comprises the formula, while the set comprising the negation of Δ unrolled to i and the assumptions Λ unrolled to i is the set of assumptions provided to the SAT solver. We call our model checking algorithm *incremental Bounded Model Checking (BMC) under assumptions*.

Some recent works dedicated to BMC propose taking advantage of look-ahead information that is available, since the instance can be unrolled beyond the current bound [12, 13]. In particular, it is proposed in [13] to apply preprocessing, including SatELite [9], for LS-based BMC, where complete look-ahead information is required to ensure soundness, as variables that are expected to appear in the future must not be eliminated. The technique of [13] cannot be applied in our application, since it is unknown a priori how the user would update the formula before subsequent invocations of the incremental model checker. The in-depth BMC algorithm of [12], which uses a limited form of look-ahead to boost BMC, served as an inspiration for our algorithm for incremental SAT solving under assumptions with step look-ahead, presented in Section 5.

3 Preprocessing under Assumptions

Preprocessing refers to a family of algorithms whose goal is to simplify the input CNF formula prior to the CDCL-based search in SAT. Preprocessing has commonly been applied in modern SAT solvers since the introduction of the SatELite preprocessor [9]. This section first explains why even a rather straightforward form of preprocessing, known as database simplification, is expected to be much more effective when used with CM as compared to LS. We then show that, unmodified, SatELite cannot be used with either CM or LS, and demonstrate how it can be modified so as to be safely used with CM.

Consider the following algorithm, which we call *database simplification* following MiniSat [8] notation: First, propagate unit clauses with Boolean Constraint Propagation (BCP). Second, eliminate satisfied clauses and falsified literals.

Database simplification is applied as an inprocessing step (that is, as an on-the-fly simplification procedure, applied at the global decision level) in modern SAT solvers [8, 14, 15]. It can be applied during preprocessing and inprocessing with either LS or CM without further modification. A key observation is that the efficiency of the first application of database simplification after a new portion

of the incremental problem becomes available can be dramatically higher when assumptions are modeled as unit clauses (as in CM) rather than as assumption literals (as in LS). Indeed, database simplification takes full advantage of unit clauses by propagating them and eliminating resulting redundancies, while it does not take any advantage of assumption literals. In addition, variables representing assumptions are eliminated by database simplification with CM, but not with LS. Our experimental data, presented in Section 6, demonstrates that database simplification eliminates significantly more clauses for CM than for LS, and that the average conflict clause length for LS is much greater than it is for CM. These two factors favor CM as compared to LS as they have a significant impact on the efficiency of BCP and the overall efficiency of SAT solving.

Consider now the preprocessing algorithm of SatELite [9]. SatELite is a highly efficient algorithm used in leading SAT solvers [8, 14, 15]. SatELite is composed of the following three techniques:

1. *Variable elimination*: for each variable v, the algorithm performs resolution between clauses containing v (denoted by V^+) and $\neg v$ (denoted by V^-). Let U be the set of resulting clauses. If the number of clauses in U is less than or equal to the number of clauses in $V^+ \cup V^-$, then the algorithm eliminates v by replacing $V^+ \cup V^-$ by U.
2. *Subsumption*: if a clause α is subsumed by the clause β, that is, $\beta \subseteq \alpha$, α is removed.
3. *Self-subsuming resolution*: if $\alpha = \alpha_1 \vee l$ and $\beta = \beta_1 \vee \neg l$, where α_1 is subsumed by β_1, then α is replaced by α_1.

It is unclear how to apply SatELite with LS, let alone make its performance efficient. It is currently unknown how to apply SatELite for incremental SAT solving, since eliminated variables may be reintroduced (unless full look-ahead information is available [13], which is not always the case). However, even if the problem of incremental SatELite is solved, it is still unclear how to efficiently propagate assumptions when SatELite is applied with LS. One cannot apply SatELite as is, since eliminating assumption literals would render the algorithm unsound. A simple solution for ensuring soundness would be *freezing* the assumption literals [4, 13], that is, not carrying out variable elimination for them. However, this solution has the same potential severe performance drawback as database simplification applied with LS as compared to CM: freezing assumptions is expected to have a significant negative impact on the preprocessor's ability to simplify the instance.

It is also unknown how SatELite can be applied with CM. The problem is that one has to keep track of pervasive and temporary clauses. Fortunately, we can propose a simple solution for this problem, based on the observation that SatELite uses nothing but resolution. SatELite can be updated as follows to keep track of pervasive and temporary clauses. If a variable is eliminated, each new clause $\alpha = \beta_1 \otimes \beta_2$ is marked as temporary iff one of the clauses β_1 or β_2 is temporary (where $\otimes$ corresponds to an application of the resolution rule). Whenever self-subsuming resolution is applied, the new clause α_1 is temporary

iff either α or β is temporary (this operation is sound since α_1 is a resolvent of α and β). No changes are required for subsumption.

4 Transforming Temporary Clauses to Pervasive Clauses

We saw in Section 3 that CM has an important advantage over LS: preprocessing is expected to be much more efficient for CM. However, LS has its own advantages. An important advantage is efficiency of learning: all the conflict clauses learned by LS are pervasive, hence they can always be reused. In CM, all the temporary conflict clauses are lost. In this section we propose an algorithm that converts temporary clauses to pervasive clauses as a post-processing step after the SAT solver is invoked. Our algorithm overcomes the above-mentioned disadvantage of CM as compared to LS.

We start by providing some resolution-related definitions. The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are also clauses, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. The resolution rule application is denoted by $\alpha_3 = \alpha_1 \otimes^v \alpha_2$. A *resolution derivation* of a target clause α from a CNF formula $G = \{\alpha_1, \alpha_2, \ldots, \alpha_q\}$ is a sequence $\pi = (\alpha_1, \alpha_2, \ldots, \alpha_q, \alpha_{q+1}, \alpha_{q+2}, \ldots, \alpha_p \equiv \alpha)$, where each clause α_i for $i \leq q$ is *initial* and α_i for $i > q$ is *derived* by applying the resolution rule to α_j and α_k, where $j, k < i$.[3] A *resolution refutation* is a resolution derivation of the empty clause $\square$. Modern SAT solvers are able to generate resolution refutations given an unsatisfiable formula.

A resolution derivation π can naturally be considered as a directed acyclic graph (dag) whose vertices correspond to all the clauses of π and in which there is an edge from a clause α_j to a clause α_i iff $\alpha_i = \alpha_j \otimes \alpha_k$ (an example of such a dag appears in Fig. 1). A clause $\beta \in \pi$ is *backward reachable* from $\gamma \in \pi$ if there is a path (of 0 or more edges) from β to γ.

Assume now that the SAT solver is invoked over the CNF formula $A = \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\} \wedge F = \{\alpha_{n+1}, \ldots, \alpha_r\}$ (where the first n clauses are temporary unit clauses corresponding to assumptions and the rest of the clauses are pervasive). Assume that the solver generated a resolution refutation π of $A \wedge F$. Let $\beta \in \pi$ be a clause. We denote by $\Gamma(\pi, \beta)$ the conjunction (set) of all the *backward reachable assumptions* from β, that is, the conjunction (set) of all the assumptions whose associated unit clauses are backward reachable from $\beta \in \pi$. Let $\Gamma(\beta)$ be short for $\Gamma(\pi, \beta)$. To transform any clause $\beta \in \pi \setminus A$ to a pervasive clause we propose applying the following operation:

$$\boxed{T2P(\beta) = \beta \vee \neg\Gamma(\beta)}$$

That is to say, we propose to update each temporary derived clause with the negations of the assumptions that were required for its derivation, while pervasive clauses are left untouched. Consider the example in Fig. 1. The proposed operation would transform α_7 to $c \vee d \vee \neg a$; α_8 to $\neg d \vee \neg b$; α_{10} to $c \vee \neg a \vee \neg b$; and α_{11} to $\neg a \vee \neg b$. The pervasive clauses α_3, α_4, α_5, α_6, and α_9 are left untouched.

[3] We force the resolution derivation to start with all the initial clauses, since such a convention is more convenient for the subsequent discussion.

Alg. 1 shows how to transform a resolution refutation π of $A \wedge F$ to a resolution derivation $T2P(\pi)$ from F, such that every clause $\beta \in \pi \setminus A$ is mapped to a clause $T2P(\beta) = \beta \vee \neg\Gamma(\beta) \in T2P(\pi)$. The pre- and post-conditions that must hold for Alg. 1 appear at the beginning of its text. The second pre-condition is not necessary, but it makes the algorithm's formulation and correctness proof easier. The algorithm's correctness is proved below.

Algorithm 1 Transform π to $T2P(\pi)$

Require: $\pi = (A = \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\}, F = \{\alpha_{n+1}, \ldots, \alpha_r\}, \alpha_{r+1}, \ldots, \alpha_p)$ is a resolution refutation of $A \wedge F$
Require: All the assumptions in A are distinct and non-contradictory
Ensure: $T2P(\pi) = (T2P(\alpha_{n+1}), T2P(\alpha_{n+2}), \ldots, T2P(\alpha_r), T2P(\alpha_{r+1}), \ldots, T2P(\alpha_p))$ is a resolution derivation from F
Ensure: For each $i \in \{n+1, n+2, \ldots, r, \ldots, p\}$: $T2P(\alpha_i) = \alpha_i \vee \neg\Gamma(\alpha_i)$
1: **for** $i \in \{n+1, n+2, \ldots, p\}$ **do**
2: **if** $\alpha_i \in F$ **then**
3: $T2P(\alpha_i) := \alpha_i$
4: **else**
5: Assume $\alpha_i = \alpha_j \otimes^v \alpha_k$
6: **if** α_j or α_k is an assumption **then**
7: Assume without limiting the generality that α_j is the assumption
8: $T2P(\alpha_i) := T2P(\alpha_k)$
9: **else**
10: $T2P(\alpha_i) := T2P(\alpha_j) \otimes^v T2P(\alpha_k)$
11: Append $T2P(\alpha_i)$ to $T2P(\pi)$

Proposition 1. *Algorithm 1 is sound, that is, its pre-conditions imply its post-conditions.*

Proof. The proof is by induction on i, starting with $i = r + 1$. Both post-conditions hold when the "for" loop condition is reached when $i = r + 1$, since $T2P(\pi)$ comprises precisely the clauses of F at that stage. Indeed, every clause α_i visited until that point is initial and is mapped to $T2P(\alpha_i) = \alpha_i$ by construction. It is left to prove that both post-conditions hold each time after a derived clause $\alpha_i \in \pi$ is translated to $T2P(\alpha_i)$ and $T2P(\alpha_i)$ is appended to $T2P(\pi)$. We divide the proof into three cases depending on the status of α_i.

When α_i is a pervasive derived clause, its sources α_j and α_k must also be pervasive by definition. By induction, we have $T2P(\alpha_j) = \alpha_j$ and $T2P(\alpha_k) = \alpha_k$, since $\Gamma(\alpha_j)$ and $\Gamma(\alpha_k)$ are empty. Hence, $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k) = \alpha_j \otimes^v \alpha_k$. Thus, it holds that $T2P(\alpha_i)$ is derived from F by resolution, so the first post-condition holds. We also have the second post-condition, since we have seen that $T2P(\alpha_i) = \alpha_j \otimes^v \alpha_k = \alpha_i$, while $\Gamma(\alpha_i)$ is empty.

Consider the case where α_i is temporary and α_j is an assumption. The second pre-condition of the algorithm ensures that α_k will not be an assumption. The algorithm's flow ensures that $T2P(\alpha_i) = T2P(\alpha_k)$. By induction, $T2P(\alpha_k)$ is derived from F by resolution, hence $T2P(\alpha_i)$ is also derived from F by resolution and the first post-condition holds. The induction hypothesis yields that

$T2P(\alpha_i) = T2P(\alpha_k) = \alpha_k \vee \neg\Gamma(\alpha_k)$. It must hold that $\alpha_k = \alpha_i \vee \neg l_j$, otherwise the resolution rule application $\alpha_i = (\alpha_j = l_j) \otimes^v \alpha_k$ would not be correct. Substituting the equation $\alpha_k = \alpha_i \vee \neg l_j$ into $T2P(\alpha_i) = \alpha_k \vee \neg\Gamma(\alpha_k)$ gives us $T2P(\alpha_i) = \alpha_i \vee \neg l_j \vee \neg\Gamma(\alpha_k) = \alpha_i \vee \neg(l_j \wedge \Gamma(\alpha_k))$. It must hold that $\Gamma(\alpha_i) = l_j \wedge \Gamma(\alpha_k)$ by resolution derivation construction. Substituting the latter equation into $T2P(\alpha_i) = \alpha_i \vee \neg(l_j \wedge \Gamma(\alpha_k))$ gives us precisely the second post-condition.

Finally consider the case where α_i is temporary and neither α_j nor α_k is an assumption. The first post-condition still holds after $T2P(\pi)$ is updated with $T2P(\alpha_i)$, since $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k)$ by construction and both $T2P(\alpha_j)$ and $T2P(\alpha_k)$ are derived from F by resolution by the induction hypothesis. The induction hypothesis yields that $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k) = (\alpha_j \vee \neg\Gamma(\alpha_j)) \otimes^v (\alpha_k \vee \neg\Gamma(\alpha_k))$. We have $\alpha_i = \alpha_j \otimes^v \alpha_k$. Hence, it holds that $T2P(\alpha_i) = (\alpha_j \otimes^v \alpha_k) \vee \neg\Gamma(\alpha_j) \vee \neg\Gamma(\alpha_k) = \alpha_i \vee \neg\Gamma(\alpha_j) \vee \neg\Gamma(\alpha_k)$. By resolution derivation construction, it holds that $\Gamma(\alpha_i) = \Gamma(\alpha_j) \wedge \Gamma(\alpha_k)$. Hence, $T2P(\alpha_i) = \alpha_i \vee \neg\Gamma(\alpha_i)$ and we have proved the second post-condition. □

We implemented our method as follows. After SAT solving is completed, we go over the derived clauses in the generated resolution refutation π and associate each derived clause α with the set $\Gamma(\alpha)$. This operation can be applied independently of the SAT solving result, even if the problem is satisfiable. After that, we update each remaining temporary conflict clause α with $\neg\Gamma(\alpha)$ and mark the resulting clause as pervasive. In practice, there is no need to create a new resolution derivation $T2P(\pi)$.

Note that one needs to store and maintain the resolution derivation in order to apply our transformation. This may have a negative impact on performance. To mitigate this problem, we store only a subset of the resolution derivation, where each clause's associated set of backward reachable assumptions is non-empty. The idea of holding and maintaining only the relevant parts of the resolution derivation was proposed and proved useful in [16].

Finally, when the number of assumptions is large, our transformation might create pervasive clauses which are too large. To cope with this problem we use a user-given threshold n. Whenever the number of backward reachable assumptions for a clause is higher than n, that clause is not transformed into a pervasive clause, and thus is not reused in subsequent SAT invocations.

5 Incremental SAT Solving under Assumptions with Step Look-Ahead

In some applications of incremental SAT solving under assumptions, look-ahead information is available. Specifically, before invocation number i, the solver may already know the clauses F_j and assumptions A_j for some or all future invocations $j > i$. In this section, we propose an algorithm for incremental SAT solving under assumptions given a limited form of look-ahead, which we call step look-ahead. The reason for choosing this form of look-ahead is inspired by step-based approaches to BMC [12].

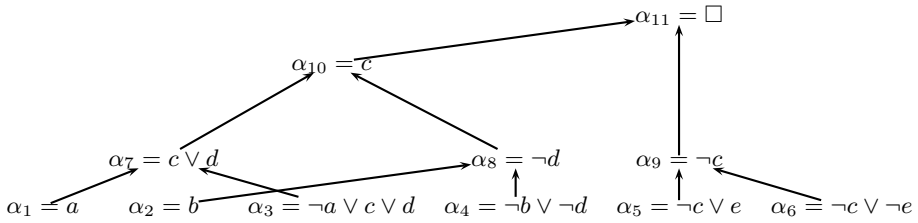

Fig. 1: An example of a resolution refutation for illustrating the *T2P* transformation. The pervasive input clauses are $F = \alpha_3 \wedge \alpha_4 \wedge \alpha_5 \wedge \alpha_6$; the assumptions are $\alpha_1 = a$ and $\alpha_2 = b$. The only pervasive derived clause is α_9; the rest of the derived clauses are temporary.

Given an integer step $s > 1$, an invocation i is *step-relevant* iff i *modulo* $s = 0$ (invocations are numbered starting with 0). Given an invocation q, its *step interval* is a set of successive invocations $SI(q) = [n * s, \ldots, q, \ldots, ((n+1) * s) - 1]$, where $n * s$ is the largest step-relevant invocation smaller than or equal to q. For example, for $s = 3$, invocations $0, 3, 6, 9, 12, \ldots$ are step-relevant; and $SI(3) = SI(4) = SI(5) = [3, 4, 5]$. In *step look-ahead*, at each step-relevant invocation i, the solver can access all the clauses and assumptions associated with invocations within $SI(i)$. In addition, in step look-ahead, given a step-relevant invocation i, it holds that $F_j \wedge A_j$ is satisfiable iff $F_j \wedge A_j \wedge F_k$ is satisfiable for every $j, k \in SI(i)$. That is to say, we assume that all the clauses available within the step interval hold for every invocation within that step interval.

One can adjust LS to take advantage of the fact that the solver has a wider view of the problem as follows. At a step-relevant invocation i, LS can be provided the problem $\bigwedge_{j=i}^{i+s-1} F_j$ and solve it s times, each time under a new set of assumptions A_j for each $j \in SI(i)$ (in this scheme non-step-relevant invocations are ignored). We call this approach the *Single instance Literal-based with Step look-ahead (LSS)* approach. LSS was proved to have advantages over the plain LS algorithm (which has a narrower view of the problem) in the context of standard BMC [12]. However, it suffers from the same major drawback as plain LS: preprocessing does not take advantage of assumptions.

We need to refine the semantics of the problem before proposing our solution. Given a step-relevant invocation i, an assumption $l \in A_i$ is *invocation-generic* iff $l \in A_j$ for every $j \in SI(i)$. Any assumption that is not invocation-generic is *invocation-specific*. That is, an assumption is invocation-generic iff it can be assumed for every invocation within the given step interval. In our application of incremental BMC under assumptions, described in Section 2, the negation of the property for each bound is invocation-specific, while the unrolled temporal assumptions are invocation-generic.

We propose an algorithm, called *Multiple instances Clause/Literal-based with Step look-ahead (CLMS)* (shown in Alg. 2), that combines LS and CM. The algorithm is applied at each step-relevant invocation. It creates the instance $\bigwedge_{j=i}^{i+s-1} F_j$ once as in LS. The key idea is that invocation-generic assumptions can be provided as unit clauses, since assuming them does not change the satisfiability status of the problem for any invocation within the current step interval.

To ensure the soundness of solving subsequent step intervals, the unit clauses corresponding to invocation-generic assumptions must be temporary as in CM. After creating the instance the solver is invoked s times for each invocation in the step interval, each time under the corresponding invocation-specific assumptions. To combine SatELite with Alg. 2 in a sound manner, all the invocation-specific assumptions must be frozen. Finally, note that our *T2P* transformation is applicable for CLMS.

Algorithm 2 CLMS Algorithm

1: **if** i is step-relevant **then**
2: Let $G = \bigcap_{j=i}^{i+s-1} A_j$ be the set of all invocation-generic assumptions
3: Create a SAT solver instance with pervasive clauses $\bigwedge_{j=i}^{i+s-1} F_j$ and temporary clauses G
4: Optionally, apply SatELite, where all the invocation-specific assumptions in $\bigcup_{j=i}^{i+s-1} A_j$ must be frozen
5: **for** $j \in \{i, i+1, \ldots, i+s-1\}$ **do**
6: Invoke the SAT solver under the assumptions $A_j \setminus G$
7: Optionally, transform temporary clauses to pervasive clauses using *T2P*
8: Store the pervasive clauses and delete the SAT instance

6 Experimental Results

This section analyzes the performance of various algorithms for incremental SAT solving under assumptions on instances generated by incremental BMC under assumptions. In our analysis, we consider an instance satisfiable iff a certain invocation over that instance by one of the algorithms under consideration was satisfiable within a time-out of one hour. We picked instances from three satisfiable families comprising satitisfiable instances only (128 instances) and four unsatisfiable families comprising unsatisfiable instances only (81 instances). We measured the number of completed incremental invocations for unsatisfiable families and the solving time until the first time an invocation was proved to be satisfiable for satisfiable families (the time-out was used as the solving time when an algorithm could not prove the satisfiability of a satisfiable instance). Each pair of invocations corresponds to a BMC bound (a clock transition and a real bound), where the complexity of SAT invocations in BMC grows exponentially with the bound. We implemented the algorithms in Intel's internal state-of-the-art Eureka SAT solver and used machines with Intel® Xeon® processors with 3Ghz CPU frequency and 32Gb of memory for the experiments.

We checked the performance of LS and CM as well as of LSS and CLMS with steps 10 and 50. We tested CM and CLMS with and without SatELite and with different thresholds for applying *T2P* transformation (0, 100, 100000). Our solver uses database minimization during inprocessing by default.

The graph on the left-hand side of Fig. 2 provides information about the number of variables and assumptions (satisfiable and unsatisfiable instances appear separately). For each instance we measured these numbers at the last invocation

completed by both CM and LS (the basic algorithms). Note that the distribution of variables and assumptions for the satisfiable instances is more diverse. This is explained by the fact that for satisfiable instances, the last invocation is sometimes very low or very high, while for unsatisfiable instances it is moderate. Overall, our satisfiable instances are easier to solve.

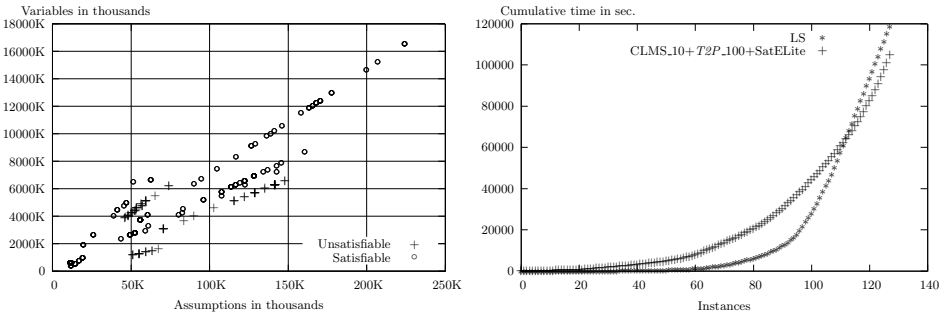

Fig. 2: Left-hand side: variables to assumptions ratio; Right-hand side: a comparison between plain LS and CLMS_10+*T2P*_100+SatELite with respect to the number of satisfiable instances solved within a given time.

Table 1: The number of invocations completed within an hour for the unsatisfiable instances from four families. The algorithms are sorted by the sum of completed invocations in decreasing order.

Algorithms				Completed Invocations				
LS?	SatELite?	Step	T2P Thr.	Overall	Fam. 1	Fam. 2	Fam. 3	Fam. 4
-	+	50	0	2967	1443	470	562	492
-	+	10	100	2934	1413	472	563	486
-	+	10	0	2932	1408	474	568	482
-	+	50	100	2927	1427	462	552	486
-	+	50	100000	2927	1427	462	552	486
-	+	1	0	2828	1365	468	539	456
-	+	1	100	2813	1363	462	535	453
-	-	10	100000	2806	1378	442	528	458
-	-	50	0	2801	1375	444	526	456
-	-	50	100	2795	1373	442	522	458
-	-	50	100000	2795	1373	442	522	458
-	-	10	100	2779	1357	440	528	454
-	-	10	0	2775	1353	438	530	454
-	-	1	100000	2736	1335	432	537	432
-	-	1	100	2734	1339	436	526	433
-	-	1	0	2732	1339	436	524	433
+	-	10	N/A	2579	1295	380	494	410
+	-	1	N/A	2575	1295	378	494	408
+	-	50	N/A	2563	1291	376	488	408
-	+	10	100000	2525	1245	390	507	383
-	+	1	100000	2250	1133	296	493	328

Consider Table 1, which compares the number of completed invocations for unsatisfiable instances. Compare basic CM and LS (configurations [-,-,1,0] and [+,-,1], respectively). CM significantly outperforms LS. As we discussed in Section 3, the reasons for this are related to the relative efficiency of database simplification and the average clause length for both algorithms. Fig. 3 demonstrates the huge difference between the two algorithms in these parameters in

favor of CM. Note that when SatELite is not applied, the best performance is achieved by CLMS_10 (CLMS with step 10) with *T2P*_100000 (*T2P* with threshold 100000). Hence, without SatELite, both CLMS and *T2P* are helpful. SatELite increases the number of completed invocations considerably, while the absolutely best result is achieved by combining SatELite with CLMS_50 when *T2P* is turned off. Fig. 4 demonstrates that the reason for the inefficiency of the combination of *T2P* and SatELite is related to the fact that the time spent in preprocessing increases drastically when *T2P* is applied with threshold 100000. The degradation still exists, but is not that critical when the threshold is 100.

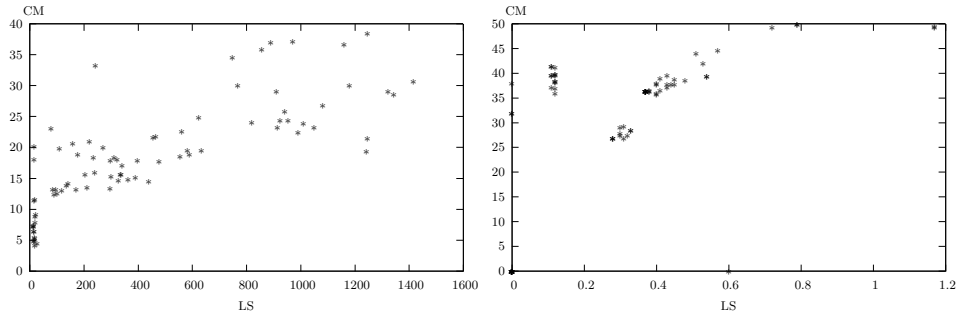

Fig. 3: Comparison of CM and LS with respect to average conflict cause length (left-hand side) and the percent of clauses removed by database simplification (right-hand side). Note the difference in the scales of the axes.

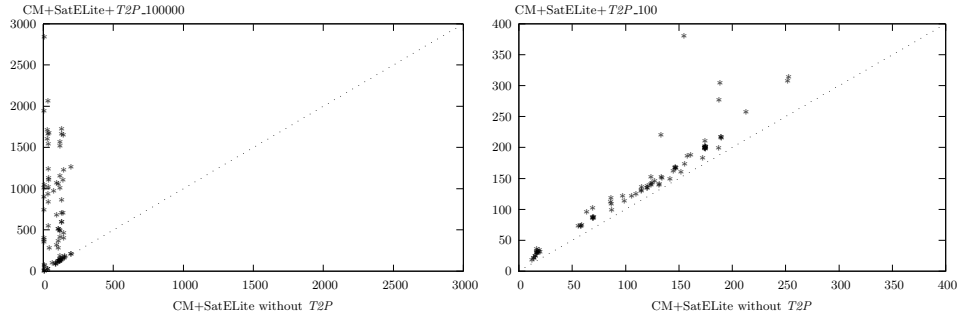

Fig. 4: Comparison between CM and CM+*T2P*_100000 (left-hand side) and between CM and CM+*T2P*_100 (right-hand side) in terms of time in seconds spent in SatELite.

Consider now Table 2, which compares the run-time for satisfiable instances. Note that, unlike in the case of unsatisfiable instances, the default LS is one of the best algorithms. The advantage of LS over CM-based algorithms is that it maintains all the information relevant to the decision heuristic. This advantage proves to be very important in the context of relatively easy falsifiable instances. Still, the absolutely best configuration is the combination of CLMS_10 with SatELite and *T2P*_100, which uses all the algorithms proposed in this paper.

The graph on the right-hand side of Fig. 2 shows that the advantage of our approach over LS becomes apparent as the run-time increases, while LS is still preferable for easier instances.

One can also see that the combination of CLMS_10 with SatELite and *T2P*_100 ([-,+,10,100]) is the most robust approach overall: it is the second best for unsatisfiable instances and the absolute best for satisfiable instances.

Table 2: Solving time in seconds for instances from three falsifiable families. The algorithms are sorted by overall solving time in increasing order.

Algorithms				Time			
LS?	SatELite?	Step	T2P Thr.	Overall	Fam. 1	Fam. 2	Fam. 3
-	+	10	100	104845	10843	35083	58919
+	-	1	N/A	118954	18005	41624	59325
-	+	10	0	134917	16886	40965	77067
+	-	10	N/A	139787	21726	53304	64757
-	+	10	100000	154437	22280	53436	78721
-	+	50	0	172104	10496	56087	105521
-	+	50	100	189965	11649	69373	108943
-	+	50	100000	192790	15220	68475	109096
-	-	10	100000	196784	12521	126153	58110
+	-	50	N/A	200261	22832	93635	83794
-	-	10	100	205124	16133	125529	63462
-	-	10	0	206390	14991	125400	65999
-	+	1	100	213278	31628	83009	98641
-	-	1	100	216714	20889	118703	77122
-	-	1	100000	220054	20639	128871	70545
-	+	1	0	219346	34447	89040	95859
-	-	1	0	228404	23642	121608	83154
-	-	50	0	244202	18996	138971	86235
-	+	1	100000	244826	34735	111862	98229
-	-	50	100000	247347	18514	138552	90281
-	-	50	100	250937	18897	141524	90516

7 Conclusion

This paper introduced efficient algorithms for incremental SAT solving under assumptions. While the currently widely-used approach (which we called LS) models assumptions as first decision variables, we proposed modeling assumptions as unit clauses. The advantage of our approach is that we allow the preprocessor to use assumptions while simplifying the formula. In particular, we demonstrated that the efficient SatELite preprocessor can easily be modified for use in our scheme, while it cannot be used with LS. Furthermore, we proposed an enhancement to our algorithm that transforms temporary clauses into pervasive clauses as a post-processing step, thus improving learning efficiency. In addition, we developed an algorithm which improves the performance further by taking advantage of a limited form of look-ahead information, which we called step look-ahead, when available. We showed that the combination of our algorithms outperforms LS on instances generated by a prominent industrial application. The empirical gap is especially significant for difficult unsatisfiable instances.

Acknowledgments

The authors would like to thank Amit Palti for supporting this work, Paul Inbar for editing the paper, and Niklas Eén, Armin Biere, and Mate Soos for their clarifications and suggestions that helped us to improve it.

References

1. Silva, J.P.M., Sakallah, K.A.: Robust search algorithms for test pattern generation. In: FTCS. (1997) 152–161
2. Strichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In Margaria, T., Melham, T.F., eds.: CHARME. Volume 2144 of Lecture Notes in Computer Science., Springer (2001) 58–70
3. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A new incremental satisfiability engine. In: DAC, ACM (2001) 542–545
4. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. **89**(4) (2003)
5. Cabodi, G., Lavagno, L., Murciano, M., Kondratyev, A., Watanabe, Y.: Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques. ACM Trans. Design Autom. Electr. Syst. **15**(2) (2010)
6. Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. [17] 121–128
7. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. [17] 181–188
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of Lecture Notes in Computer Science., Springer (2003) 502–518
9. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of Lecture Notes in Computer Science., Springer (2005) 61–75
10. Khasidashvili, Z., Kaiss, D., Bustan, D.: A compositional theory for post-reboot observational equivalence checking of hardware. In: FMCAD, IEEE (2009) 136–143
11. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC. (1999) 317–320
12. Khasidashvili, Z., Nadel, A.: Implicative simultaneous satisfiability and applications. In: HVC'11 (to appear). (2011)
13. Kupferschmid, S., Lewis, M.D.T., Schubert, T., Becker, B.: Incremental preprocessing methods for use in BMC. Formal Methods in System Design **39**(2) (2011) 185–204
14. Biere, A.: Lingeling and Plingeling. `http://fmv.jku.at/lingeling/`.
15. Soos, M.: Cryptominisat2. `http://www.msoos.org/cryptominisat2`.
16. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In Sakallah, K.A., Simon, L., eds.: SAT. Volume 6695 of Lecture Notes in Computer Science., Springer (2011) 174–187
17. Bloem, R., Sharygina, N., eds.: Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23. In Bloem, R., Sharygina, N., eds.: FMCAD, IEEE (2010)