Ultimately Incremental SAT

Alexander Nadel [1] Vadim Ryvchin[1,2] Ofer Strichman[2]
alexander.nadel@intel.com rvadim@tx.technion.ac.il
ofers@ie.technion.ac.il

[1] Design Technology Solutions Group, Intel Corporation, Haifa, Israel
[2] Information Systems Engineering, IE, Technion, Haifa, Israel

Abstract. Incremental SAT solving under assumptions, introduced in Minisat, is in wide use. However, Minisat's algorithm for incremental SAT solving under assumptions has two main drawbacks which hinder performance considerably. First, it is not compliant with the highly effective and commonly used preprocessor SatELite. Second, all the assumptions are left in the formula, rather than being represented as unit clauses, propagated, and eliminated. Two previous attempts to overcome these problems solve either the first or the second of them, but not both. This paper remedies this situation by proposing a comprehensive solution for incremental SAT solving under assumptions, where SatELite is applied and all the assumptions are propagated. Our algorithm outperforms existing approaches over publicly available instances generated by a prominent industrial application in hardware validation.

1 Introduction

Modern backtrack search-based SAT solvers are indispensable in a broad variety of applications [2]. In the classical SAT interface, the solver receives one formula in Conjunctive Normal Form (CNF) and is required to decide whether it is satisfiable or unsatisfiable. However, many practical applications [18–20, 10, 4, 11, 8] require solving a sequence of related SAT formulas.

Rather than solving related formulas separately, modern solvers attempt to solve them *incrementally*, that is, to propagate information gathered during the solving process to future instances in the sequence. Initially there was *Clause Sharing (CS)* [19, 20], in which relevant learned clauses are propagated between instances. Then came Minisat [9], whose widely-used incremental interface induces the problem of *incremental SAT solving under assumptions*. In incremental SAT solving under assumptions the user may invoke the solving function multiple times, each time with a different set of *assumption literals* and, possibly, additional clauses. The solver then checks the satisfiability of all the clauses provided so far, while enforcing the values of the current assumptions only. This paper aims to improve the algorithm to solve the problem of incremental SAT solving under assumptions.

In Minisat's approach to solving the problem, the same SAT solver instance solves the entire sequence internally, and hence the state of the solver is preserved between incremental steps, which means that it shares not only the relevant learned clauses, but also the scores that guide decision, restart, and clause-deletion heuristics. The assumptions are modeled as first decision variables; all inferred clauses that depend on some of the assumptions include their negation, which means that they are *pervasive*, i.e., they are implied by the formula regardless of the assumptions and can therefore be reused in subsequent steps. We refer to this approach as *Minisat-Alg*.

Independently of advances in incremental SAT solving, a breakthrough in *non-incremental* SAT solving was achieved with the SatELite [7] preprocessor. SatELite applies non-increasing *variable elimination* (existentially quantifying variables by resolution as long as this does not increase the size of the formula), subsumption and self-subsumption until fixed-point, before the SAT solver begins the search. Currently, SatELite is the most dominant and useful preprocessing algorithm [1].

Combining Minisat-Alg with SatELite could clearly make incremental SAT solving faster. Unfortunately, however, these two methods are incompatible. The problem is that one cannot eliminate a variable that may reappear in future steps. In [15] we suggested a solution to this problem. We will briefly explain this solution, which we call *incremental SatELite*, in Section 2.2.

Another problem with combining Minisat-Alg and preprocessing relates to the assumptions. Minisat-Alg uses assumptions as first decision variables and hence cannot enjoy the benefits of modeling them as unit clauses. This is necessary for forcing the inferred clauses to be pervasive. However, modeling them instead as unit clauses typically leads to major simplifications of the formula with BCP and SatELite, and this can have a crucial impact on the size of the formula and the performance of the solver.

In [13] the first two authors suggested a method that we call here *assumption propagation*, which enjoys both worlds, at the price of using multiple SAT instances. It models assumptions as units, but then it transforms *temporary* (i.e., assumption-dependent) clauses to pervasive clauses after each invocation using the so-called *Temporary2Pervasive (T2P)* transformation, i.e., the assumptions upon which each inferred clause depends are added to it, negated. Assumption propagation can be combined with SatELite [13]. Assumption propagation is inspired by CS, which would use multiple SAT instances, encode assumptions as units, and reuse the set of pervasive conflict clauses to solve our problem. CS, however, is neither compliant with SatELite nor does it transform temporary clauses to pervasive.

Our new approach, which we call *Ultimately Incremental SAT (UI-SAT)*, combines the advantages of the approaches we have presented. We summarize the differences between UI-SAT and the other algorithms in Table 1. The columns represent the following:

1. *Algorithm:* the algorithm name and origin.
2. *Instances:* the number of SAT instances used. Recall that a single instance is preferable because of score sharing.

3. *Assumptions as units:* indicates whether assumptions are modeled as unit clauses, which, via SatELite and BCP, simplifies the formula.
4. *SatELite:* indicates the way SatELite is applied. CS and Minisat-Alg cannot apply SatELite at all, while assumption propagation applies full SatELite for each incremental step. Incremental SatELite and UI-SAT carry out a more efficient procedure while enjoying the same effect: they apply SatELite incrementally, which means that they handle only the newly-provided clauses and assumptions for each step.
5. *Assumption-dependent clauses:* the way in which assumption-dependent conflict clauses are treated. CS marks all the conflict clauses that depend on assumptions and discards them before the next incremental step. Potentially useful information is lost this way. Minisat-Alg and incremental SatELite reuse all such clauses across all the incremental steps, where the negation of the assumption literals are part of the clauses. Assumption propagation operates similarly to CS, except that it applies *T2P*. Such an approach allows one to reuse the conflict clauses for the next step instead of discarding them, while still enjoying the benefits of modeling assumptions as unit clauses. Finally, UI-SAT introduces an incremental version of *T2P*: before step i it only adds a literal $\neg l$ if l was an assumption literal in step $i-1$ and is not an assumption at step i.

Algorithm	*Instances*	*Assumptions as units*	*SatELite*	*Assumption-dep. Clauses*
Clause Sharing (CS) [19, 20]	Multiple	Yes	No	Discard
Minisat-Alg [9]	One	No	No	Keep all
Assumption prop.[13]	Multiple	Yes	Full	*T2P*
Incremental SatELite [15]	One	No	Incremental	Keep all
UI-SAT (this paper)	One	Yes	Incremental	Incremental *T2P*

Table 1. A comparison of different approaches to incremental SAT solving under assumptions.

As is clearly evident from the table, our new approach, UI-SAT, is the most comprehensive solution to the problem of incremental SAT solving under assumptions to date. Note that we solve the most general problem. First, we do not make any a priori suppositions about the use of assumptions. The assumption literals can be part of the original problem and/or be selector variables; they can be flipped between instances or discarded, and new assumption literals can be introduced. Second, we do not assume that look-ahead information is available. Indeed, in various applications information about future instances is not available:

– Some applications require interactive communication with the user for determining the next portion of the problem:

    - One example is an article from IBM [5] that shows that using an *incremental* SAT-based model checker, based on an incremental SAT solver, is critical for speeding-up regression verication, where a new version of a hardware design is re-verified with respect to the same (or very similar) specication. The changes in the design and the specification are not known a-priory.
    - The benchmarks used in this paper spring from another such application in formal verification. 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 [12]. The engineer then invokes an *incremental* bounded model checker 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 counterexample. 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 other applications the calculation of the next portion of the problem depends on the results of the previous invocation of the SAT solver. For example, various tasks in MicroCode validation [11] are solved by using a symbolic execution engine to explore the paths of the program. The generated proof obligations are solved by an incremental SAT-based SMT solver. In this application, the next explored path of the program is determined based on the result of the previous computation.

In the experimental results section, we show the superiority of UI-SAT to the other algorithms over the same instances used in [13]; these were generated by Intel's incremental bounded model checker and are publicly available.

We continue in the following section with formalization and various preliminaries, which are mostly similar to those in [15]. In Section 3 we present our algorithm, and in Section 4 we summarize our empirical evaluation. We conclude in Section 5.

2 Preliminaries

Let φ be a CNF formula. We denote by $vars(\varphi)$ the variables used in φ. For a clause c we write $c \in \varphi$ to denote that c is a clause in φ. For $v \in vars(\varphi)$ we define $\varphi_v = \{c \mid c \in \varphi \wedge v \in c\}$ and $\varphi_{\bar{v}} = \{c \mid c \in \varphi \wedge \bar{v} \in c\}$ (a somewhat abusing notation, as we refer here to v as both a variable and a literal). When a literal is given as an assumption and in a future step it stops being an assumption, we say that it is *invalidated*.

At step 0 of incremental SAT solving under assumptions the SAT solver is given a CNF formula φ^0 and a set of assumption literals A_0. It has to solve φ^0 under A_0, i.e., to determine whether $\varphi^0 \wedge A_0$ is satisfiable. At each step i for $i > 0$, additional clauses Δ^i and a different set of assumption literals A_i are provided to the solver. We denote by φ^i all the input clauses available at step i: $\varphi^i \equiv \varphi^0 \wedge \bigwedge_{j=1..i} \Delta^j$. At step i, the solver should solve $\varphi^i \wedge A_i$.

2.1 Preprocessing

The three preprocessing techniques that are implemented in SatELite are:

Variable elimination
Input: formula φ and a variable $v \in vars(\varphi)$.
Output: formula φ' such that $v \notin vars(\varphi')$ and φ' and φ are equisatisfiable. Typically this technique is applied for a variable v only if the number of clauses in φ' is not larger than in φ.

Subsumption
Input: $\varphi \wedge (l_1 \vee \cdots \vee l_i) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.
Output: $\varphi \wedge (l_1 \vee \cdots \vee l_i)$.

Self-subsumption
Input: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j \vee \bar{l})$.
Output: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.

2.2 Preprocessing in an Incremental Setting

The problem in combining variable elimination and incremental SAT is that a variable that is eliminated at step i can later reappear in Δ^j for some $j > i$. Since the elimination at step i may remove *original* clauses that contain that variable, soundness is possibly lost. For example, suppose that a formula contains the two clauses $(v_1 \vee v), (v_2 \vee \bar{v})$. Eliminating v results in removing these two clauses and adding the resolvent $(v_1 \vee v_2)$. Suppose, now, that in the next iteration the clauses $(\bar{v}), (\bar{v_1})$ are added, which clearly contradict $(v_1 \vee v)$. Yet since we erased that clause and since there is no contradiction between the resolvent and the new clauses, the new formula is satisfiable.

The solution we proposed in [15], which is also the basis for the generalization we propose in the next section, appears in pseudo-code in Alg. 1. The algorithm is applied at the beginning of each incremental step. Consider the pseudo-code. *SubsumptionQ* is a global queue of clauses.

For each $c \in SubsumptionQ$, and each $c' \in \varphi^i$, RemoveSubsumptions executes these two conditional steps:

Algorithm 1 Preprocessing in an incremental SAT setting (without assumptions propagation), as it appeared in [15].

```
1: function PREPROCESS-INC(int $i$)                              ▷ preprocessing of $\varphi^i$
2:     $SubsumptionQ = \{c \mid \exists v.\ v \in c \wedge v \in vars(\Delta^i)\}$;
3:     REMOVESUBSUMPTIONS ();
4:     for ($j = 0 \ldots |ElimVarQ| - 1$) do          ▷ scanning eliminated vars in order
5:         $v = ElimVarQ[j].v$;
6:         if $|\varphi_v^i| = |\varphi_{\bar{v}}^i| = 0$ then continue;
7:         REELIMINATE-OR-REINTRODUCE($j$, $i$);
8:     while $SubsumptionQ \neq \emptyset$ do
9:         for each variable $v \notin ElimVarQ$ do                      ▷ scanning the rest
10:            $SubsumptionQ =$ ELIMINATEVAR-INC($v$, $i$);
11:            REMOVESUBSUMPTIONS ();
12:        $SubsumptionQ = \{c \mid vars(c) \cap TouchedVars \neq \emptyset\}$;
13:        $TouchedVars = \emptyset$;
```

1. if $c \subset c'$ it performs subsumption;
2. else if c self-subsumes c' then it performs self-subsumption.

The rest of the algorithm relies on information that we maintain as part of the process of variable elimination. $ElimVarQ$ is a queue of pairs $\langle v, num \rangle$ where v is a variable that was eliminated (or reeliminated as we explain below) in step $i - 1$, and num is the number of resolvent clauses resulting from eliminating v (this number is later used for deciding whether to reintroduce v). In addition, we save the original clauses that are removed when eliminating v in sets S_v and $S_{\bar{v}}$ according to $v's$ phase in those clauses. If v later reappears in some Δ^i, we either *reeliminate* or *reintroduce* v based on these sets. These sets do not contain *all* the original clauses that contain v, because some of them might have been removed from the formula earlier, when another variable, which shares a clause with v was eliminated. We showed in [15] that as long as the order in which the variables are reeliminated or reintroduced remains fixed, soundness is preserved. This order is indeed enforced in lines 4–7.

We now explain how REELIMINATE-OR-REINTRODUCE works by returning to the example above. Suppose that v was eliminated in the base iteration and v_1 was not. We have $S_v = (v_1 \vee v)$, $S_{\bar{v}} = (v_2 \vee \bar{v})$, and $\Delta^1 = \{(\bar{v}), (\bar{v_1})\}$. We can now decide to do one of the following for each variable that is currently eliminated (only v in our case):

– Reeliminate v. For this we need to compute the resolvents that would have been generated if v was not already eliminated. Let $\varphi_v^1 = \{\}$ and $\varphi_{\bar{v}}^1 = \{(\bar{v})\}$ denote the subset of clauses in φ^1 that contain v and $\bar{v}$, respectively.[1] We

[1] These sets may contain not only new clauses from Δ^i as in this example, but also clauses that were retrieved when variables earlier-in-the-order were reintroduced.

compute

$$\begin{aligned}&\textsc{Resolve}(\varphi_v^1, \varphi_{\bar{v}}^1) \cup \textsc{Resolve}(\varphi_v^1, S_{\bar{v}}) \cup \textsc{Resolve}(S_v, \varphi_{\bar{v}}^1) = \\ &\{\} \ \cup \ \{\} \ \cup \ \textsc{Resolve}((v_1 \vee v), (\bar{v})) = (v_1) \ ,\end{aligned}$$

add it to the formula and remove all clauses containing v or $\bar{v}$. This leaves us with $(v_1 \vee v_2) \wedge (v_1) = (v_1)$. Now adding $(\bar{v_1})$ leads to the desired contradiction.

– Reintroduce v. For this we only need to add $S_v, S_{\bar{v}}$ back to the formula, which leaves us with $(v_1 \vee v_2), (v_1 \vee v), (v_2 \vee \bar{v}), (\bar{v}), (\bar{v_1})$, which is again contradictory, as it should be.

A reasonable criterion for choosing between these two options is the expected size of the resulting formula after this step.

Variables not in $ElimVarQ$ are preprocessed in the loop starting in line 8. It is basically the same procedure that appears in SatELite, iterating between variable elimination and subsumption, with a small difference in the implementation of the variable elimination, as can be seen in EliminateVar-inc of Alg. 2: in addition to elimination, it also updates $ElimVarQ$ and the S sets. We have ignored in this short description various optimizations and subtleties related to difference between original and conflict clauses. A correctness proof of this algorithm can be found in a technical report [16].

Algorithm 2 Variable elimination for φ^i, where the eliminated variable v was *not* eliminated in φ^{i-1}.

```
1: function EliminateVar-inc(var $v$, int $i$)
2:     clauseset $Res$ = Resolve $(\varphi_v^i, \varphi_{\bar{v}}^i)$;        ▷ Resolve all. Remove tautologies.
3:     if $|Res| > |\varphi_v^i| + |\varphi_{\bar{v}}^i|$ then return $\emptyset$;        ▷ no variable elimination
4:     $S_v = \varphi_v^i$; $S_{\bar{v}} = \varphi_{\bar{v}}^i$;        ▷ Save for possible reintroduction
5:     $ElimVarQ$.push($\langle v, |Res|\rangle$);        ▷ Save #resolvents in queue
6:     $\varphi^i = (\varphi^i \cup Res) \setminus (\varphi_v^i \cup \varphi_{\bar{v}}^i)$;
7:     ClearDataStructures $(v)$;
8:     $TouchedVars = TouchedVars \cup vars(Res)$;        ▷ used in Alg. 1
9:     return $Res$;
```

3 Adding Assumption Propagation

We now describe an incremental version of the preprocessing algorithm that supports assumption propagation. As mentioned in Section 1, we are targeting the general case, where assumptions may appear in both polarities in the formula, and where in each increment the set of assumption variables that is used is arbitrary, e.g., Δ^i may include assumption variables that have been used before. Assuming that the algorithm described in Sec. 2.2 is correct (as was proven in [16]), to maintain correctness while adding assumption propagation we maintain three invariants about the formula ψ being solved at step i:

1. $\varphi^i \implies \psi$. Specifically, this implies that every conflict clause that was learned in previous instances and appears in ψ is implied by φ^i.
2. All the original clauses of φ^i that were subsumed in previous iterations owing to assumptions that are now invalidated, appear in ψ.
3. All the literals in the original clauses of φ^i, which were self-subsumed in previous iterations owing to assumptions that are now invalidated, are restored to their original clauses in ψ.

We say that a clause is *temporary* if it is either an assumption or was derived from one or more assumptions, and *pervasive* otherwise. We mark a conflict clause as temporary during conflict analysis if one of its antecedents is marked that way. For each temporary clause c we maintain a list $SubsumedClauses[c]$ of clauses that were subsumed by c. As can be seen in Alg. 3, our subsumption function is similar to that of SatELite [7], with the difference that we update $SubsumedClauses[c]$ with the subsumed clause in line 6. This allows us to retrieve these clauses in future instances, in which the root assumptions of c are no longer valid (i.e., the assumptions upon which c depends).

Algorithm 3 Eliminating the subsumed clauses, and saving on the side those that depend on assumptions (line 6). Without this line this algorithm is equivalent to that used in SatELite. In future instances where some of the root assumptions of c are not valid, we retrieve the clauses from $SubsumedClauses[c]$.

```
1: function SUBSUME(Clause $c$)
2:     Pick the literal $p$ in $c$ that has the shortest occur list;
3:     for each $c' \in occur(p)$ do                 ▷ $occur(p) = \{c \mid p \in c, c \in \varphi\}$
4:         if $c$ is a subset of $c'$ then
5:             Remove $c'$ from the clauses database;
6:             if $c$ is temporary then $SubsumedClauses[c].Add(c')$;
```

self-subsumption is presented in Alg. 4. Whereas in SatELite self-subsumption is done simply by erasing a literal, here we distinguish between temporary and pervasive clauses. For pervasive clauses we indeed erase the literal (line 9), but for temporary clauses we perform resolution. This results in adding the same clause as in the case of pervasive clauses (c'', which is added in line 8, is equivalent to the subsumed clause c' minus the literal p), but this resolution is recorded explicitly in the resolution DAG, and consequently we are able to update $SubsumedClauses[c'']$ with the subsumed clause c'.

The net result of the modifications to the SUBSUME and SELFSUBSUME functions is that we have a resolution DAG with all the information needed to undo the effect of an assumption literal, hence maintaining the three invariants. Indeed, let us now shift our focus to UNDOASSUMPTIONS in Alg. 5. This function computes, for each clause c, the set $A(c)$ of invalidated assumption literals upon which it depends. Our implementation does so via BFS traversal of the resolution graph starting from each assumption literal in $A_{i-1} \setminus A_i$. It then performs

Algorithm 4 Self-subsumption. We perform resolution (rather than just eliminate a literal as in SatELite) and save the eliminated clause, because we will need to retrieve it in future instances in which some of the root assumptions of c are invalidated.

```
1: function selfSubsume(Clause c)
2:     for each $p \in c$ do
3:         for each $c'$ subsumed by $c[p := \bar{p}]$ do
4:             if $c$ is temporary then
5:                 $c'' = res(c, c')$;
6:                 $SubsumedClauses[c].Add(c')$;
7:                 Remove $c'$ from clause database;
8:                 AddClause($c''$);
9:             else remove $\bar{p}$ from $c'$;
```

T2P in line 7, namely it adds to c the negation of the literals in $A(c)$, so it can be reused regardless of the values of those assumption literals, and hence invariant #1 is maintained. Note that this is an improvement over [13] because it is incremental: it only adds those assumption literals that were invalidated at the current step, whereas in [13] the entire set of assumptions upon which the clause depends is recomputed each time.

Algorithm 5 Undoing the effect of invalidated assumptions. Line 5 refers also to deleted clauses. In line 7 we do not consider a clause c if it has been deleted (this is a heuristic decision).

```
1: function UndoAssumptions
2:     For each clause $c$, $A(c) = \emptyset$;
3:     $RootAssumptions$ = assumptions in $A_{i-1} \setminus A_i$;
4:     for each $a \in RootAssumptions$ do
5:         for each clause $c$ descendant of $a$ on the resolution graph do
6:             $A(c) = A(c) \cup \{a\}$;
7:     for each clause $c$ for which $A(c) \neq \emptyset$ do
8:         Replace $c$ with $(\left(\bigvee_{a \in A(c)} \bar{a}\right) \vee c)$;          ▷ T2P
9:         for each clause $c' \in SubsumedClauses[c]$ do addClause($c'$);
```

In addition, in line 9 the function restores the clauses that were subsumed by c, and hence maintains invariant #2 and, indirectly, also invariant #3 because of the way self-subsumption is done, as was shown in Alg. 4. There are several subtleties in implementation of this algorithm that are not mentioned explicitly in the pseudo-code. First, the clauses visited in line 5 also include learned clauses that have been deleted, because such clauses may have subsumed other clauses, and may have participated in inferring other (temporary) clauses. Hence, when the deletion strategy of the solver deletes a learned clause, we do not remove it from the resolution DAG. Whether or not to turn such a deleted clause into a pervasive one in line 7 is a matter of heuristics. In our implementation we do not. Finally, observe that line 8 replaces each unit clause c corresponding to an assumption literal $a \in \mathit{RootAssumptions}$ with a tautology $(\bar{a} \vee a)$. Our implementation therefore simply erases such clauses.

Given this ability to undo the effect of assumptions, we can now integrate this function with the PREPROCESS-INC function of Alg. 1. Consider PREPROCESS-INC-WITH-ASSUMPTIONS in Alg. 6. It undoes the effect of the invalidated assumptions by calling UNDOASSUMPTIONS, adds as unit clauses the new assumptions in A_i, and finally calls PREPROCESS-INC. This is the algorithm that we call at the beginning of each new instance.

Note that, unlike in Minisat-Alg, CS, and incremental SatELite, one needs to store and maintain the resolution derivation in order to implement our algorithm. This may have a negative impact on performance. To mitigate this problem, we store only a partial resolution derivation that is sufficient to implement our approach: only clauses that are backward reachable from the assumptions are stored. Using partial resolution was initially proposed in [17] in the context of unsatisfiable core extraction. It was first used in the context of incremental SAT solving under assumptions in the assumption propagation algorithm [13].

Algorithm 6 Preprocessing in an incremental SAT setting with assumptions propagation.

```
1: function PREPROCESS-INC-WITH-ASSUMPTIONS(int $i$)        ▷ preprocessing of $\varphi^i$
2:     UNDOASSUMPTIONS();
3:     $\forall a \in A_i \setminus A_{i-1}$ ADDCLAUSE($(a)$);        ▷ Assumptions as units;
4:     PREPROCESS-INC($i$);        ▷ See Alg. 1
```

4 Experimental Results

This section analyzes the performance of our new algorithm UI-SAT against the existing algorithms for incremental SAT solving under assumptions listed in Table 1 in Section 1.

We used instances generated by Intel's formal verification flow applied to the *incremental* formal verification of hardware, where the model checker is allowed to be re-used incrementally each time with a different set of assumptions. Such a mode is essential for ensuring fast and effective application of formal verification in industrial settings. Further details about the application are provided in Section 2 of [13]. We used the same instance set as in [13]. Overall, 210 instances were used, but the results below refer only to the 186 of them that were fully solved by at least one solver within the time limit of 3600 seconds. The instances are characterized by a large number of assumptions, all of which being part of the original formula (and not selector variables). The assumptions mimic the environment of a sub-circuit that is verified with incremental BMC in Intel, as described in Section 1. All of the instances are publicly available at [14].

	CS	Minisat-Alg	Assump. Prop.	Incr. SatELite	UI-SAT
Run-time	223,424	159,423	182,530	209,781	**64,176**
Unsolved	28	14	24	16	**1**

Table 2. Performance results (the run-time is in seconds).

We implemented all the algorithms in Intel's new SAT solver Fiver. Note that the implementation in [13] was made in another SAT solver (Eureka), hence the results presented in this paper are not directly comparable to the results presented in [13].

For the experiments we used machines running Intel® Xeon® processors with 3Ghz CPU frequency and 32Gb of memory.

Table 2 and Figure 1 present the results of all the algorithms. Note that Minisat-Alg refers to Minisat's algorithm as implemented in Fiver, rather than to the Minisat SAT solver.

UI-SAT is clearly the fastest approach. UI-SAT outperforms the next best algorithm Minisat-Alg by 2.5x and solves 13 more instances. CS is the slowest algorithm, as expected. Interestingly, Minisat-Alg outperforms both assumption propagation and incremental SatELite. In [13] assumption propagation is faster than Minisat-Alg in the presence of step look-ahead (that is, a limited form of look-ahead information), while incremental SatELite is faster than Minisat-Alg in [15] on a different instance set having considerably fewer assumptions. One can see that the integration of incremental SatELite and assumption propagation is critical for performance.

Figure 2 provides a direct comparison between UI-SAT and Minisat-Alg. One can see that UI-SAT outperforms Minisat-Alg on the majority of difficult instances. Overall, UI-SAT is faster than Minisat-Alg by at least 2 sec. on 110 instances, while Minisat-Alg is faster than UI-SAT by at least 2 sec. on 58 instances. However, when the run-time of both solvers is at least 100 sec., the difference in performance is more substantial, as UI-SAT is faster than Minisat-Alg on 107 instances, while Minisat-Alg is faster than UI-SAT on only 17 instances.

5 Conclusion

We have introduced a new algorithm for incremental SAT solving under assumptions, called Ultimately Incremental SAT (UI-SAT). UI-SAT ensures that SatELite is applied and all the assumptions propagated, that is, represented as unit clauses. We have shown that UI-SAT outperforms existing approaches, including clause sharing, Minisat's algorithm, incremental SatELite (without assumption propagation), and assumption propagation (without incremental SatELite) on a publicly available set of 186 instances generated by an incremental bounded model checker. UI-SAT is 2.5x faster than the second best approach, and solves 13 more instances.

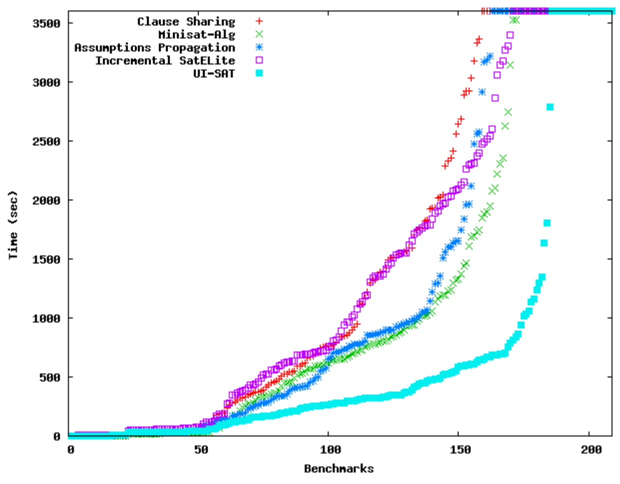

Fig. 1. A comparison of the different techniques over 186 industrial instances generated by a bounded model checker of hardware.

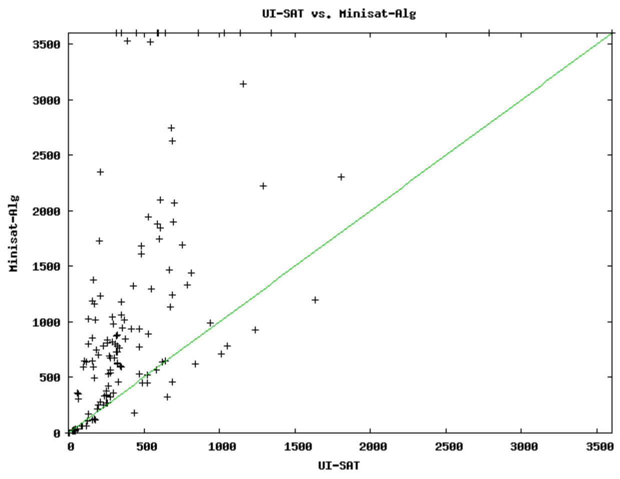

Fig. 2. Comparing Minisat-Alg to UI-SAT.

Acknowledgments

The authors would like to thank Daher Kaiss for supporting this work and Paul Inbar for editing the paper.

References

1. Adrian Balint and Norbert Manthey. Boosting the performance of SLS and CDCL solvers by preprocessor tuning. In *Pragmatics of SAT*, 2013.
2. Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
3. Roderick Bloem and Natasha Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. IEEE, 2010.
4. Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques. *ACM Trans. Design Autom. Electr. Syst.*, 15(2), 2010.
5. Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 135–143. FMCAD Inc., 2011.
6. Alessandro Cimatti and Roberto Sebastiani, editors. *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*. Springer, 2012.
7. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
8. Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In Bloem and Sharygina [3], pages 181–188.
9. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
10. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
11. Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In Bloem and Sharygina [3], pages 121–128.
12. Zurab Khasidashvili, Daher Kaiss, and Doron Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143. IEEE, 2009.
13. Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Cimatti and Sebastiani [6], pages 242–255.
14. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. UI-SAT benchmark set: `https://copy.com/osV4myggyNRa`.
15. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. In Cimatti and Sebastiani [6], pages 256–269.

16. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. Technical Report IE/IS-2012-02, Technion, 2012. Available also from `http://ie.technion.ac.il/~ofers/publications/sat12t.pdf`.
17. Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *SAT'11*, number 6695 in LNCS, pages 174–187, 2011.
18. João P. Marques Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *FTCS*, pages 152–161, 1997.
19. Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.
20. Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.