Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 119 (2005) 51–65
www.elsevier.com/locate/entcs

An Incremental Algorithm to Check Satisfiability for Bounded Model Checking [1]

HoonSang Jin[2] Fabio Somenzi[3]

University of Colorado at Boulder

Abstract

In Bounded Model Checking (BMC), the search for counterexamples of increasing lengths is translated into a sequence of satisfiability (SAT) checks. It is natural to try to exploit the similarity of these SAT instances by *forwarding* clauses learned during conflict analysis from one instance to the next. The methods proposed to identify clauses that remain valid fall into two categories: Those that are oblivious to the mechanism that generates the sequence of SAT instances and those that rely on it. In the case of a BMC run, it was observed by Strichman [20] that those clauses learned during one SAT check that only depend on the structure of the model remain valid when checking for longer counterexamples. Eén and Sörensson [9] pointed out that all learned clauses can be forwarded if the translation into SAT obeys commonly followed rules. Many clauses that are forwarded this way, however, are of little usefulness and may degrade performance. This paper presents an extension to Strichman's approach in the form of a more general criterion to filter conflict clauses that can be profitably forwarded to successive instances. This criterion, in particular, is still *syntactic* and quite efficient, but accounts for the presence of both *primary* and *auxiliary objectives* in the SAT instance. This paper also introduces a technique to *distill* clauses to be forwarded even though they fail the syntactic check. Distillation is a *semantic* approach that can be applied in general to incremental SAT, and often produces clauses that are independent of the primary objective, and hence remain valid for the remainder of the sequence of instances. In addition, distillation often improves the quality of the clauses, that is, their ability to prevent the examination of large regions of the search space. Experimental results obtained with the CirCUs SAT solver confirm the efficacy of the proposed techniques, especially for large, hard problems.

Keywords: bounded model checking, propositional satisfiability, conflict-learned clauses, incremental algorithms.

[1] This work was supported in part by SRC contract 2003-TJ-920.

[2] Email: Jinh@Colorado.EDU

[3] Email: Fabio@Colorado.EDU

doi:10.1016/j.entcs.2004.06.062

1 Introduction

Bounded Model Checking (BMC) [3] determines whether for model $\mathcal{K}$ there exists a counterexample to property φ of length less than or equal to k. If such a counterexample is found, or if k is large enough [19,18,6,1], then BMC effectively answers the question $\mathcal{K} \models \varphi$; otherwise, it increases the user's confidence in the correctness of $\mathcal{K}$.

BMC converts the search for a counterexample of a certain maximum length into a sequence of propositional satisfiability (SAT) checks. In its simplest form, the length starts from 0 and is incremented by 1 for each instance. At the i-th step of this iteration, a propositional formula is built from $\mathcal{K}$ and φ that is satisfiable if and only if there exists a counterexample to φ in $\mathcal{K}$ of length $k = i - 1$. Though variations on this scheme are easy to envisage, and can be accommodated by the techniques discussed in this paper, we shall limit our discussion to this case. Each formula checked for satisfiability consists of three parts, corresponding to the initial state constraint, the unrolled transition relation, and the property to be satisfied. In the last part, one often distinguishes a *primary objective* (e.g., the last state of the counterexample violates an invariant) from *auxiliary objectives* (e.g., no state except the last one violates that invariant). Auxiliary objectives express information about the problem gathered from failed attempts to find shorter counterexamples. They may help in directing the search process.

The emergence of efficient SAT solvers over the last decade [21,25,26,11] has greatly contributed to the success of BMC. The new generation of SAT solvers improves over the classical DPLL procedure [8,7] in several ways. Of interest to us are conflict analysis and clause recording: When a conflicting assignment is found, it is analyzed to identify a subset that is still conflicting. The disjunction of the negation of the literals in the subset is a *conflict-learned clause* (or, more concisely, a *conflict clause*) that can be added to the given SAT instance to prevent the examination of regions of the search space that are guaranteed to contain no solutions. Not all conflict clauses are worth keeping; many SAT solvers periodically discard those that have proved ineffective.

Incremental SAT solvers [24,20,9] try to leverage the similarity between the elements of a sequence of SAT instances; most do so by re-utilizing conflict clauses, though when many closely related instances must be solved, caching solutions [15] and incremental translation [2] can also be effective. If a SAT instance is obtained from another by adding some clauses (as in [12]), then all conflict clauses of the first can be *forwarded* to the second. This is correct because the second instance implies the first, which in turn implies all its (conflict) clauses. Therefore, when clauses are only added through the sequence of instances, there is no need to screen conflict clauses to determine which ones can be forwarded. This, on the other hand, is necessary when arbitrary clauses are both added and subtracted to create a new instance. A common approach to such general case is to have the incremental SAT solver keep track of whether a conflict clause depends on some removed clauses. The approach of [24] is to record, for each conflict clause, the clauses that made up the corresponding implication graph. This approach does not require any foreknowledge of the subsequent SAT instances to be solved incrementally, and does not restrict the changes possible from one instance to the next; however, keeping track of dependencies may be expensive.

Strichman [20] was the first to observe that in BMC some clauses are known to survive through all instances in the sequence. A formula passed by BMC to the SAT solver contains clauses that describe the transition relation of the model unrolled a number of times. These clauses are not discarded when the length of the counterexample is increased. Hence, a conflict clause that depends only on them can be forwarded. The advantage of this approach is that complete dependence information is no longer needed; one-bit marker per clause is sufficient. Such a marker is derived from the structure of the implication graph that produces the clause. Therefore, we speak of a *syntactic* criterion in this case.

The authors of [9] remarked that tracking dependencies is not required if only *unit* clauses are removed. Such clauses can be regarded as assumptions by the SAT solver. As a result, a conflict clause incorporates its assumptions or some of their implied literals, and is not invalidated when the assumptions are repealed. It was further observed in [9] that the standard encoding of objectives into SAT guarantees that the clauses that must be removed when the counterexample length is incremented are unit clauses. Hence, all conflict clauses can be forwarded. The approach of [9] exemplifies those incremental satisfiability algorithms that are aware of the mechanism generating the sequence of SAT instances. On the other hand, when one of its unit clause assumptions is reversed, a conflict clause becomes satisfied and therefore inert.

Having many inert clauses in the solver may significantly affect performance. Therefore we want to forward only clauses that have a good chance of remaining active in successive instances. To this purpose, we propose a *syntactic* criterion that improves on the one of [20] in two ways. First, it accounts for *auxiliary objectives*, and hence can forward more clauses. Second, it does not require the examination of the entire implication graph when marking a conflict clause.

We also present a *semantic* forwarding criterion, which *distills* the clauses that cannot be forwarded according to the syntactic check into clauses implied by the new instance. These distilled clauses are sometimes independent of the objective of the new instance and usually more effective than the raw clauses from which they are derived in preventing exploration of fruitless regions of the search space.

The rest of this paper is organized as follows. Section 2 reviews background material. Section 3 describes the incremental SAT algorithm, while Section 4 discusses the experiments conducted to assess its effectiveness. Section 5 concludes.

2 Preliminaries

Let $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$ be sets of Boolean variables. We designate by V' the set $\{v'_1, \ldots, v'_n\}$ consisting of the primed version of the elements of V, and by V^i the set $\{v^i_1, \ldots, v^i_n\}$. Likewise, $W^i = \{w^i_1, \ldots, w^i_m\}$. An *open system* is a 4-tuple

$$\Omega = \langle V, W, I, T \rangle \enspace ,$$

where V is the set of (current) state variables, W is the set of combinational variables, $I(V)$ is the initial state predicate, and $T(V, W, V')$ is the transition relation. The variables in V' are the next state variables. All sets are finite, and all variables range over finite domains.

Bounded Model Checking (BMC) [3] reduces the search for a counterexample to a linear time property to propositional satisfiability. Given an open system Ω, an LTL [17] formula φ, and a bound k, BMC tries to refute $\Omega \models \varphi$ by proving the existence of a witness of length k to the negation of the LTL formula.

BMC generates a propositional formula $[\![\Omega, \neg\varphi]\!]_k$ that is satisfiable if and only if a counterexample to φ of length k exists in Ω; $[\![\Omega, \neg\varphi]\!]_k$ is defined as follows:

$$[\![\Omega, \neg\varphi]\!]_k = I(V^0) \wedge \bigwedge_{0 \leq i < k} T(V^i, W^i, V^{i+1}) \wedge [\![\neg\varphi]\!]_k \enspace , \tag{1}$$

where $[\![\neg\varphi]\!]_k$ expresses the satisfaction of $\neg\varphi$ along that path. (See [3] for the details of the translation.) It is customary to write $[\![\neg\varphi]\!]_k$ as $\omega_k \wedge F_k$, where ω_k is a literal called the *primary objective*. If it is known that $[\![\Omega, \neg\varphi]\!]_j$ is unsatisfiable for $j < k$, then one can conjoin (1) with

$$\bigwedge_{0 \leq i < k} \neg[\![\neg\varphi]\!]_i \enspace . \tag{2}$$

Each term $\neg[\![\neg\varphi]\!]_i$ is written $\neg\omega_i \wedge F_i$, where $\neg\omega_i$ is an *auxiliary objective*.

```
DPLL() {
    while (ChooseNextAssignment())
        while (Deduce() == CONFLICT) {
            blevel = AnalyzeConflict();
            if (blevel $\leq$ 0) return UNSATISFIABLE;
            else Backtrack(blevel);
        }
    return SATISFIABLE;
}
```

Fig. 1. DPLL algorithm with conflict analysis

A SAT solver returns assignments to the variables of a propositional formula that satisfy it, if such assignments exist. A *literal* is either a variable or its complement, a *clause* is a disjunction of literals from distinct variables, and a *conjunctive normal form* (CNF) formula is a conjunction of clauses. An *And-Inverter Graph* (AIG) [16] is a Boolean circuit such that each node's output is the conjunction of its two inputs. The arcs of the circuit may be *complementing*. A Binary Decision Diagram (BDD) [5] is a Boolean circuit such that each node is a multiplexer controlled by an input variable. Most SAT solvers operate on a propositional formula in CNF. Our SAT solver CirCUs [14,13], on the other hand, accepts a combination of CNF clauses, AIG, and reduced, ordered BDDs. Each result of a conflict analysis is represented as one clause [10]. Hence, the algorithms described in this paper can be applied to any SAT solver based on clause recording.

Figure 1 shows the pseudocode of the DPLL procedure as implemented in most modern SAT solvers, including CirCUs. The algorithm maintains a list of assignments that is initialized with the unit clauses from the SAT instance. If all variables have been given a value, a satisfying set of assignments has been found. Otherwise, an assignment is either extracted from the list, or created by a new decision; it is added to the assignment stack, and its consequences are deduced. Every time a new decision is made, the *decision level*, which is initially 0, is incremented. If a conflict is detected, it is analyzed. The results of the analysis are a conflict clause and a backtracking decision level. The latter tells how much of the assignment stack should be erased (decreasing the decision level) before continuing the search.

Conflict analysis relies on the *implication graph*, which is a directed acyclic graph (DAG) whose nodes are the variables in the current set of assignments plus a special *conflict node* if the assignments are conflicting. The arcs of the DAG are such that if the predecessors of node ν are $\nu_1, \ldots, \nu_m$, then there exists a clause, an AIG node, or a BDD, such that it implies the value of ν given the values of $\nu_1, \ldots, \nu_m$. The predecessors of the conflict node identify a clause, AIG node, or BDD, whose assignments are inconsistent. A root of the graph corresponds to a decision assignment. Note that different implication graphs may be obtained from the same set of assignments, depending on the order in which their implications are propagated. A conflict clause is obtained by disjoining the negation of the literals forming a cut in the implication graph that separates the conflict node from the roots of the graph. The First Unique Implication Point (UIP) approach [26] starts from the conflict node and looks for the first cut such that it contains exactly one literal implied by the most recent decision.

Every non-root node of the implication graph identifies an *antecedent* clause: The implied value of the node contributes one literal, and the negation of the predecessor values supplies the others. Some of these clauses correspond to clauses in the input description or were derived from previous conflicts. Others come from AIG nodes or BDDs. For instance, an AIG node $a = b \wedge c$, and assignment $a = 1$ implying $c = 1$ implicitly give the clause $(\neg a \vee c)$. The conflict clause is obtained by successive resolutions starting from the conflicting clause associated to the conflict node. At each step one literal implied at the current decision level is resolved using its antecedent clause. All the clauses involved in the resolution are implied by the function whose satisfiability is being checked.

3 Forwarding Clauses

We consider an incremental SAT algorithm that exploits the similarities among SAT instances that form a sequence by using the conflict clauses generated from the previous instances to guide the search for a solution to the current instance. We assume that the second and successive instances of the sequence are obtained by removing some clauses from the instances immediately preceding them, and then adding some other clauses.

In BMC the unsatisfiability of a SAT problem usually comes from the simultaneous constraining of the initial and final state of a path because the formula representing the unrolled transition relation and the constraint on the initial states is normally satisfiable. However, this does not mean that the conflicts the solver goes through in proving unsatisfiability involve variables from most time frames. First, there may be conflicts due to inconsistent assignments to the inputs and outputs of some circuit elements. Second, the proof of unsatisfiability may rely on conflicts that establish non-trivial facts about intermediate states of possible counterexamples, given the constraints on the initial states. Figure 2 provides some intuition for how *local* conflicts arise. It shows an AIG produced by unrolling a transition relation twice. The property being checked is an invariant. The three parts of the figure are three snapshots taken during the search. Each circle is a node of the AIG. A circle is filled if the node is assigned a value. The three snapshots suggest

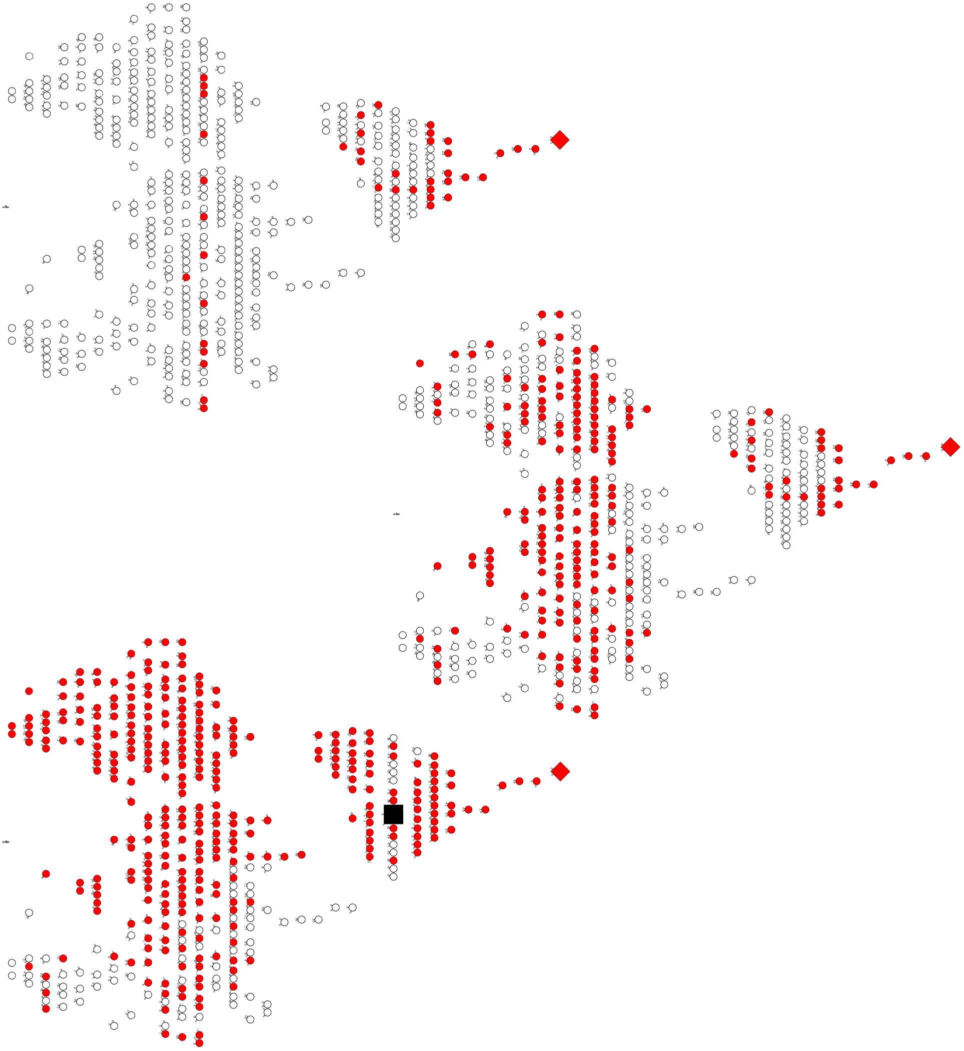

Fig. 2. Examples of justification clouds

that the implication graph initially consists of several connected components. If a conflict occurs when extending one of the components not including the objective (the diamond at the far right), then the resulting conflict clause is totally independent of the current objective and is a good candidate for forwarding.

As recalled in Section 1, it was noted in [9] that when objectives are identified by literals, all conflict clauses can be forwarded. However, a clause that contains the old primary objective, is trivially satisfied when that objective

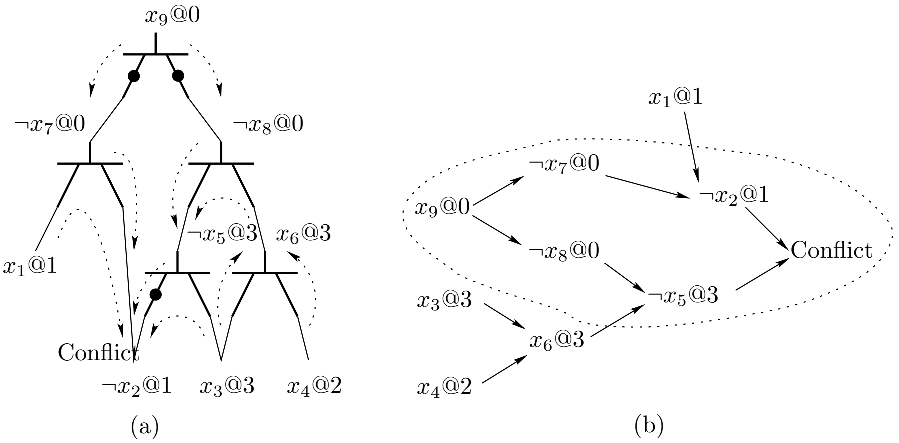

Fig. 3. Example of tracing objective

is turned into auxiliary by negating its literal. In general, the usefulness of conflict clauses that depend on the primary objective is dubious, even when they do not contain the objective literal. Hence, in the following, we propose two techniques to identify *objective-independent* clauses.

3.1 Objective Tracing

We are interested in extending the criterion of [20] to account for auxiliary objectives since they contribute to many conflicts, especially when looking for looping counterexamples.

Definition 3.1 Let $[\![\neg\varphi]\!]_k = \omega_k \wedge F_k$. A conflict clause γ is *objective-dependent* if ω_k is an ancestor of the conflict node in the implication graph, or at least one objective-dependent clauses is used in its resolution. Otherwise γ is *objective-independent.*

We show an AIG and an implication graph for it in Figure 3. Each horizontal line in Figure 3(a) represents an AIG node; a dot stands for complementation. The objective is x_9 and a conflict happens after three decisions have been made for x_1, x_4, and x_3. Along with the implications, we also propagate the objective flag through the implication graph. For example, we mark x_7 and x_8 because they are implied by the objective. The dotted line in Figure 3(b) encloses the marked nodes.

In our incremental algorithm, conflict analysis has the additional goal to check whether the conflict is related to the objective. The conflict in Figure 3 is objective-dependent, since one of the ancestors of the conflict node is x_9. A naive approach could identify objective-dependent conflicts by checking whether the objective is in the transitive fanin of the conflict node. However, most modern SAT solvers, including CirCUs, use the first UIP to find concise explanations of conflicts. Therefore, standard conflict analysis does not need to traverse all the transitive fanin of the conflict node. Hence the naive approach may incur overhead.

Since we propagate the objective flag during implication, we can check if the conflict is related to the objective by checking the mark of the conflict node. If the conflict node is not marked then we need to traverse the implication graph to check whether objective-dependent conflict clauses are the reason for the current conflict. However, we only traverse until the first UIP is found. Even though the rest of the implication graph has objective-dependent conflict clauses, they can be ignored. The reason of the conflict is isolated from those clauses by the first UIP.

In [20], the author propose a method to identify conflict clauses to be forwarded in BMC based solely on the circuit structure. First, all the clauses created from the circuit structure are marked. Once a conflict happens, one checks if all clauses leading to the conflict are marked. It so, the conflict is derived from inconsistency between the current assignment and the circuit structure. Therefore, the conflict is marked for forwarding. This method does not account for auxiliary objectives, which, as shown in Section 4, often speed-up BMC. Second when BMC is applied to optimized circuits, in which most redundancies have been removed, the clauses that are solely derived from the circuit structure tend to be few. This occurs in our experimental setup, since we apply BDD sweeping [16] to remove redundancy.

3.2 Distillation

Although the criterion of Section 3.1 forwards more clauses than the one of [20], it is still rather conservative and may miss many useful conflict clauses. Therefore, in this section, we develop a *semantic* criterion that is applied to small clauses that failed the syntactic check based on dependency on the objective.

To distill a clause under the new objective, we check whether the clause is satisfied under the assignments that are implied by the unit clauses of the new SAT instance. If the clause is satisfied, it is discarded. Otherwise, we assert the negations of its literals and carry out the resulting implications. If this does not result in a conflict, the clause is discarded. (Therefore, distillation can be applied also when not all clauses can be forwarded.) Otherwise, the clause obtained by conflict analysis is the distilled version of the given clause and is forwarded.

Even though we limit the number of literals in the candidate clauses, there

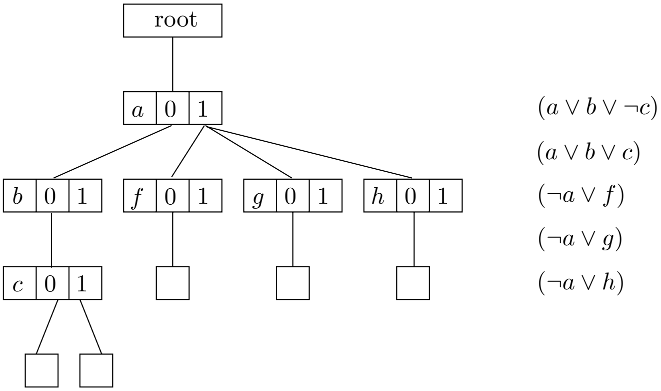

Fig. 4. Example of clause trie. Each node has two sets of children corresponding to the two literals of each variables.

may still be many of them. Therefore, we build a trie (cf. [25]) with the set of candidates. Figure 4 shows an example. With the trie, we can distill the clauses with at most 7 decisions, instead of the 12 decisions required if we process the clauses one by one. We do not explicitly optimize the trie when clauses can be merged. In Figure 4, $(a \vee b \vee \neg c)$ and $(a \vee b \vee c)$ could be merged into $(a \vee b)$. If the new clause indeed causes a conflict, it will be found when trying $(a \vee b \vee \neg c)$. The size of the trie depends on the order of the variables. We sort the variables according to their number of occurrences in the candidate clauses.

Figure 5 shows the distillation algorithm. For each element in the trie, the decision on the children '0' and '1' is made in DistillationAux() if there is a non-empty suffix from them. Conflict analysis is invoked when BCP() results in a conflict. If the resulting conflict clause has fewer literals than the number of trie nodes on the path from the root, it is forwarded. Otherwise the conflict clause based on the decisions that have been made is forwarded. The former case is more frequent. Since we want to go through all the trie nodes one by one, we use chronological backtracking based on the trie structure.

The distillation process has several advantages. First, it is a semantic approach that may derive clauses that were not produced by previous SAT checks. Second, some of these clauses are reusable because they do not depend on the current objective. Since the criterion based on tracing the objective is conservative, we often find many objective-independent clauses from the objective-dependent clauses of the previous run. Third, the quality of conflict clauses usually is improved by distillation. This is partly due to the different order in which assignments are made, and which results from the traversal of the trie. Moreover, the first UIP tends to be closer to the conflict than the literals in the clause to be distilled that it replaces.

```
Distillation ($Trie$) {
    for each $t$ in $Trie$ {
        if (Value($t.node$)! $= X$) {
            Distillation($t.child$[Value(($t.node$)]);
            continue ;
        }
        DistillationAux($t, 0$);
        DistillationAux($t, 1$);
    }
}

DistillationAux ($t, value$) {
    if ($t.child[value]$) {
        $level$ = MakeDecisionBasedOnTrie($t.node, value$);
        if (BCP($level$) == $CONFLICT$) {
            $conflictClause$ = ConflictAnalysis($level$);
            if (numLiterals($conflictClause$) < $level$)
                AddConflictClause($conflictClause$) ;
            else
                AddConflictCaluseBasedOnTrieNode();
            UndoImplication($level$);
            return ;
        }
        else {
            Distillation($t.child[value]$);
        }
    }
}
```

Fig. 5. Distillation algorithm

A final, important advantage of distillation is that the variable scores used to make decisions are updated during the process. Therefore, distillation biases the search based on information from the previous instances in the sequence. In [23], the entire proof of unsatisfiability from one SAT run is used to bias the variable scores of the next run. With distillation, only the part of the proof that is still useful with the new primary objective affects the scores.

4 Experimental Results

We have implemented the clause forwarding techniques in CirCUs, which is built on top of VIS-2.1 [4,22]. To show the effectiveness of objective tracing and distillation, we compare non-incremental SAT to incremental SAT. The non-incremental version of CirCUs was shown to be competitive with a popular CNF SAT solver, Zchaff, in [14].

The experimental setup is as follows. We build BMC instances for given LTL properties from the VIS benchmark suite [22]. We check for counterexamples of length up to 20. We first expand the AIG for the prescribed number of time frames and then apply BDD sweeping [16] to the result to remove redundancy. The maximum number of literals of a clause that undergoes distillation is 50.

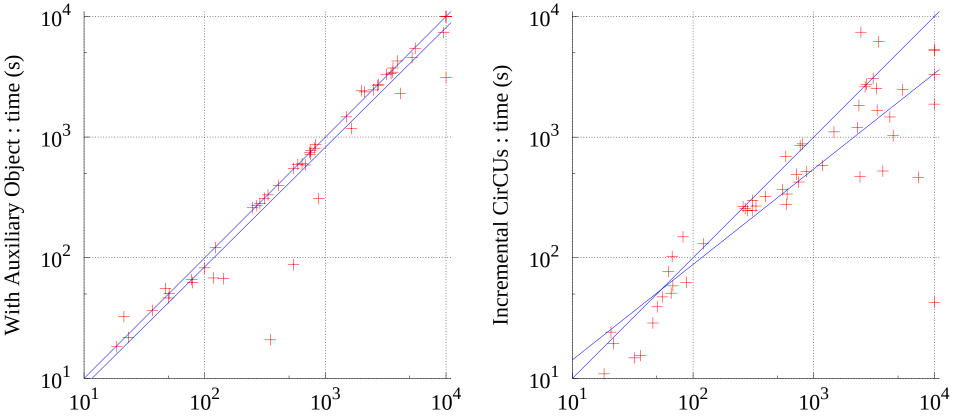

Fig. 6. Effects of auxiliary objective (left) and effects of incremental solution (right)

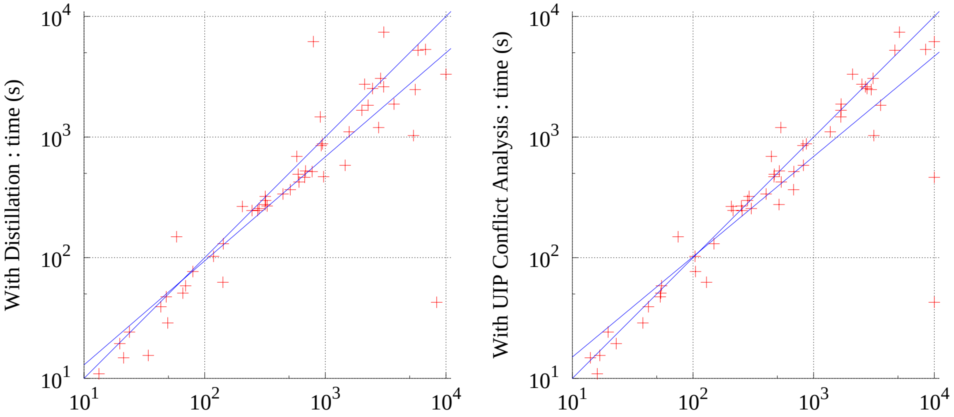

Fig. 7. Effects of distillation (left) and effects of UIP conflict analysis (right)

All experiments have been performed on a 1.7 GHz Pentium IV with 1 GB of RAM running Linux. We have set the time-out limit to 10000 s. Two lines are drawn in each plot: the main diagonal, and the result of least square fitting for $y = a \cdot x^b$.

The left scatter plot of Figure 6 shows that using auxiliary objectives provides a rather consistent speed-up. The combined effect of all the new techniques, including auxiliary objectives, objective tracing, and distillation,

Table 1
Number of reused conflict clauses by various methods

				Distillation		Lit/Conflict	
Design	total	[20]	Tracing	non-obj.	obj.	before	after
simple8	19350	76	353	2014	2954	5.7	5.0
cups	38292	720	1891	5663	11222	5.7	5.1
blackjack	43852	335	619	1883	2765	4.5	4.0
gcd	54383	437	1013	4246	4446	5.2	4.8
two	68255	422	979	2890	6340	6.0	5.4
vending	106898	361	1144	5083	7720	5.2	4.6
goodbakery	118323	268	636	4053	9074	5.3	4.6
rcnum16	122263	591	1530	5567	21939	6.5	5.8
spinner32	154788	298	3205	12924	33181	5.5	5.1
all	206398	248	1458	4253	7303	5.9	5.3

is shown in the right plot of Figure 6 by comparison to non-incremental SAT.

To show the impact of distillation, we compare incremental SAT with and without distillation in the left plot of Figure 7. To justify the claim that UIP-based conflict analysis enhances the quality of conflict clauses we compare it to using the clauses as they come out of the trie. As one can see in the right plot of Figure 7, the use of UIP-based conflict analysis often generates better results.

The number of conflict clauses forwarded by several methods is shown in Table 1. The number of forwarded clauses by the method of [20] is shown in the third column. It is collected from BMC runs for all timeframes. The fourth column shows the number of forwarded clauses by the proposed objective tracing method. This method identifies many more clauses than the method of [20].

The fifth and sixth columns show the number of clauses forwarded by distillation. The objective-independent clauses from distillation are collected from all timeframes, but the objective-dependent clauses from distillation are generated from the last timeframe only. Only the objective-independent clauses are forwarded to the next runs.

To support the claim that the quality of conflict clauses is improved by distillation, we show the number of literals per conflict. We achieve a reduction of approximately 10% in the number of literals per conflict.

5 Conclusions

We have presented two techniques for efficient incremental SAT checking in BMC. One is a syntactic technique that identifies clauses that can be profitably forwarded from one SAT instance to the next by tracing their dependence on the primary objective of the SAT problem. The other technique distills clauses that fail the tracing criterion into fewer, smaller clauses that can be forwarded. Experiments indicate that the combination of these two techniques greatly increases the number of forwarded clauses over previous methods, while preventing many useless clause from cluttering the solver's data structures. This results in a significant improvement in the speed of BMC. Though we have described our techniques for a hybrid solver used for BMC, they are applicable in general to solvers based on clause recording, and to problems that benefit from an incremental approach to satisfiability.

References

[1] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*. Springer-Verlag, Berlin, July 2004. To appear.

[2] M. Benedetti and S. Bernardini. Incremental compilation-to-SAT procedures. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[4] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[6] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 85–96, Venice, Italy, Jan. 2004. Springer. LNCS 2937.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[9] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[10] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the Design Automation Conference*, pages 747–750, New Orleans, LA, June 2002.

[11] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[12] J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1–2):177–186, Jan. 1993.

[13] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*. Springer-Verlag, Berlin, July 2004. To appear.

[14] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.

[15] J. Kim, J. Whittemore, J. P. M. Silva, and K. A. Sakallah. On solving stack-based incremental satisfiability problems. In *Proceedings of the International Conference on Computer Design*, pages 379–382, Sept. 2000.

[16] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference*, pages 232–237, Las Vegas, NV, June 2001.

[17] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Jan. 1985.

[18] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[19] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 108–125. Springer-Verlag, Nov. 2000. LNCS 1954.

[20] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *Correct Hardware Design and Verification Methods (CHARME 2001)*, pages 58–70, Livingston, Scotland, Sept. 2001. Springer. LNCS 2144.

[21] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[22] URL: http://vlsi.colorado.edu/∼vis.

[23] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proceedings of the Design Automation Conference*, pages 535–538, San Diego, CA, June 2004.

[24] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, Las Vegas, NV, June 2001.

[25] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.