Incremental Theorem Proving

S. Andrei, A. Cheng, G. Grigoras, L. Osborne

Abstract. *Theorem proving is a challenging task for formal verification of systems. There exist many efforts to efficiently solve this problem, based for example on rewriting rules and/or SAT-based techniques. We propose an alternative of SAT-based techniques by using instead a counting SAT-based technique (denoted also #SAT). A SAT solver tests if a propositional formula F has at least one truth assignment, while a #SAT solver returns the number of truth assignments of F. For efficiency reasons, many of the existing SAT-based techniques are applied incrementally, that is, using the satisfiability of some sub-formulas to determine the satisfiability of a given formula. While there exist incremental SAT solvers, to the best of our knowledge, our paper presents first time the theoretical background for the incremental counting satisfiability problem. Being a more general technique than the existing works, our approach can be used to handle all the problems solved by SAT solvers. Moreover, our #SAT solver outperforms a SAT solver when considering the challeging problems of re-design or debugging of systems.*

1. Introduction

Theorem proving is a challenge task useful for formal verification of the correctness of large systems. Since 50's, mathematicians started implementing theorem provers by constructing truth tables for statements in propositional logic. During its long and impressive history, there were many efforts to solve theorem proving efficiently and automatically, including Herbrand universe [DaP60], resolution [Rob65], T-unification [Plo72], structural induction [BMS75], term rewriting [Ble77], binary decision diagrams [Bry86], timely resolution [AbM86], and SAT-based techniques [Hor98]. There is still a competition between these known methods as well as a need for new efficient techniques. Given a particular class of systems, a method may work better than others. For example, considering systems expressed in modal logics, it seems that SAT-based decision procedures are more efficient than decision procedures based on translation methods [GGST00].

In the effort to get the benefits of all known techniques, it seems that the best way is to combine some of them. For small state space systems, model checking represents a better alternative for establishing the correctness. Thus, many theorem proving systems integrate model checkers as decision procedure. The first model checker based on BDDs (Binary Decision Diagrams, [Bry86]) was SMV [smv]. A reimplementation and extension of SMV is NuSMV, which was designed to be an open architecture for model checking. Recently, NuSMV2 combines the BDD-based model checking component and a SAT-based model checking component.

Similarly, „smvsat" [smvsat02] is a symbolic model checker which includes bounded model checking, proof-based abstraction, and SAT-based model checking. The „smvsat" is a proof generating SAT solver meaning that in the unsatisfiable case it produces a proof of the empty clause using resolution steps.

A similar professional tool is CVC (Cooperating Validity Checker) [BDBGS04], which decides logical validity of quantifier-free formulas in classical first-order logic with equality, enriched with certain background theories. The background theories are for linear real arithmetic, arrays, and inductive datatypes. For propositional reasoning,CVC incorporates Chaff [MMZZM2001], a state-of-the-art SAT solver. Moreover, CVC can emit independently checkable proofs of valid formulas.

Another industrial theorem prover is ICS (Integrated Canonizer and Solver, [ics05]). ICS is a standalone decision procedure for a combination of several theories. It incorporates a SAT solver, and is able to discharge propositionally complex formulas over the decided theories.

One more impressive theorem prover is Thunder [AFFHPV05], a bounded model checker based on a SAT technique. Thunder is useful to verify complex temporal properties on large RTL designs. The novel idea there was the introduction of the explicit induction, which makes the induction scheme an explicit part of the specification.

The motivation of our paper is to introduce a new technique based on incremental counting SAT, as an alternative for the SAT-based technique. We show that our technique may outperform the SAT-based approaches, when considering the redesign and debugging of systems. To the best of our knowledge, this paper presents for the first time the theoretical background of incremental counting SAT technique. Our experiments [AnC04] show that this is a promising technique as an alternative for SAT approaches.

2. Incrementality for SAT and Counting SAT Problems

The SAT problem *(Given a propositional formula F, is there a truth assignment for F?)* was the first discovered $\mathcal{NP}$*-complete* problem [Coo71]. Stephen Cook proved that any decision problem $P \in \mathcal{NP}$ can be polynomially transformed to the SAT problem. The SAT problem can be used to solve other problems, like for example the famous open problem „ $\mathcal{P}$ versus $\mathcal{NP}$ ", or equivalently the inclusion $\mathcal{NP} \subseteq \mathcal{P}$ or $\mathcal{NP} \not\subseteq \mathcal{P}$ .This important problem was formulated independently by Stephen Cook and Leonid Levin in 1971 [Coo71, Lev73]. The original paper [Lev73] may be hard to find, but its English translation as well as many interesting stories about the subject can be found in [Tra84]. The Clay Mathematics Institute included this problem ($\mathcal{P}$versus $\mathcal{NP}$) in its list of Millennium Prize Problems and have designated a prize of $1 milion [Cla2000].

Given a boolean formula F, an algorithm using the satisfiability of some sub-formulas of F to determine the satisfiability of F, is said to be *incremental*. Such incremental algorithms are efficient because when checking the satisfiability of F, only the provided final results for the sub-formulas of F are used, and not the recomputation of the whole F. The basic incremental satisfiability problem of propositional logic has been introduced in [Hoo93] as follows: *„Given a propositional formula F, check whether $F \cup \{C\}$ is satisfiable for a given clause C"*. The algorithm presented in [Hoo93] solves the SAT problem using the Davis-Logemann-Loveland's procedure [DLL62] combined with a backtracking strategy that adds one clause at a time. For example, a SAT solver able to handle non-conjunctive normal form constraints and incremental satisfiability was presented in [WKS01].

As stated in [GaJ79, Pap94], the counting (enumeration) problems are another type of interesting problems, but they might be intractable even if $\mathcal{P}=\mathcal{NP}$. It is still unknown whether these problems can be solved at any fixed level of the polynomial-time hierarchy [Sto77]. A counting problem P determines how many solutions exist, not just an answer „Yes/No" like a decision problem. The same concept of completeness can be defined for counting problems. A counting problem P is in $\#\mathcal{P}$ if there is a non-deterministic algorithm such that for each instance I the number of guesses that lead to the acceptance of I is exactly the number of distinct solutions of P regarding I and such that the length of the longest accepting computation is bounded by a polynomial in the length of I [Val79]. The *$\#\mathcal{P}$-complete* problems are at least as hard as *$\mathcal{NP}$-complete* problems, but probably much harder. The problem of counting the number of truth assignments (denoted by #SAT) is: *Given a propositional formula F, how many truth assignments exist for F?*. It was proved to be *$\#\mathcal{P}$-complete* [Sim75, Val79]. Obviously, an algorithm for solving #SAT problem can also solve the SAT problem. An important subclass of propositional formulas was described in [CrH96] by a #SAT Dichotomy Theorem. It was shown that if all logical relations used in generalized #SAT are affine, then the number of truth assignments can be computed in polynomial time, otherwise the problem is $\#\mathcal{P}$-complete [CrH96].

There exist already algorithms for solving the #SAT problem [Iwa89, Dub91, Tan91, And95, Zha96, And04]. In this paper we are addressing to a more challenging problem, namely the incremental #SAT problem: *„Knowing the number of truth assignments of F, what is the number of truth assignments of $F \cup \{C\}$, for any arbitrary clause C?"*. To be best of our knowledge, there is no similar work dealing with this particular problem. Because our algorithm deals with the incremental side of the problem, this is definitely more efficient than existing algorithms, so there is no need to count again the number of truth assignments for the whole F. Our investigation was occasioned by efforts to solve timing constraints verification for real-time systems [JaM86, JaM87, AnC04, ACCL05], but it has application whenever one wishes to check again for logical inferences after enlarging a propositional knowledge base. In fact, the technique described in this paper is useful not only for verification and debugging of real-time systems but for any kind of systems in general.

We now briefly sketch how to model re-design and debugging problems using incremental counting satisfiability in comparison with any arbitrary SAT solver. Usually, to prove that a formula ϕ is a theorem, a common technique [ChL73] is to show that its negation (that is, $\neg \phi$) is unsatisfiable. Most of the techniques reduce to show that a corresponding propositional formula F is unsatisfiable. If $\neg \phi$ is not unsatisfiable, then re-design and debugging approaches make sense to be considered. These approaches refer to do some allowed changes (additions or removals) in $\neg \phi$ such that the new formula $\neg \phi'$ becomes unsatisfiable. Of course, the changes in $\neg \phi$ correspond to some changes into the propositional formula F. Let us denote with $SF_1, \ldots, SF_k$, the new propositional formulas which can be added and with UF the unchanged subformula of F. The goal is to check which of formulas $UF \cup SF_1, \ldots, UF \cup SF_k$ is unsatisfiable (if none of them is unsatisfiable, we need to continue the process of changing clauses), without re-computing the satisfiability of UF. In this way, we are dealing with a real incremental approach. The number of truth assignments of $UF \cup SF_1, \ldots, UF \cup SF_k$ guides to the decision of which formula to choose: *the one with minimum number of truth assignments*. Note that this information is not provided by any existing SAT solver. Instead, our approach benefits of a sort of *monotonocity*, that is, once the formula gets new subformulas over the same alphabet of propositional variables, then the number of truth assignments is decreasing (details in Section 5).

Section 3 presents useful concepts and notations. Section 4 describes the main results for solving the incremental counting satisfiability problem. The efficiency of our incremental #SAT approach against a classical SAT approach is presented in Section 5. The conclusions and references end this paper.

3. Preliminaries

In this section we introduce some concepts, examples and notations [And95, And04] to allow the text to be self contained. For a finite set A, $|A|$ denotes the number of elements of A. The number of all sets with i elements from a set with n elements is denoted by $\binom{n}{i}$, and it is equal to $\frac{n!}{(n-i)!i!}$, where $n!=2\cdot 3\cdot \ldots n$.

Let $\mathbb{LP}$ be the *propositional logic* over the finite set of *atomic formulae* (known also as *propositional variables)* denoted by $V = \{A_1, A_2, \ldots, A_n\}$. A *literal* L can be an atomic formula A *(positive* literal), or its negation $\neg A$ *(negative* literal). We put $\overline{L} = \neg A$ if $L = A$ and $\overline{L} = A$ if $L = \neg A$ and we denote $V(L) = V(\overline{L}) = A$. Any function $S: V \to \{0,1\}$ is a *structure* (known also as assignment, substitution, instance, or model) and it can be uniquely extended in $\mathbb{LP}$ to F (this extension will be denoted also by S). The binary vector $(y_1, \ldots, y_n)$ is a truth assignment for F over $V = \{A_1, \ldots, A_n\}$ iff $S(F) = 1$ such that $S(A_i) = y_i, \forall i \in \{1, \ldots, n\}$. A formula F is called *tautology* iff for any structure S, it follows that $S(F)=1$. A formula F is called *satisfiable* iff there exists a structure S for which $S(F) = 1$. A formula F is called *unsatisfiable* (or *contradiction)* iff F is not satisfiable. Any propositional formulae $F \in \mathbb{LP}$ can be translated into the *conjunctive normal form* (CNF): $F = (L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{l,1} \vee \ldots \vee L_{l,n_l})$, where

$L_{i,j}$ are literals.In this paper, we shall use a set representation $F = \{\{L_{1,1},..., L_{1,n}\}, ...,\{L_{l,1}, ...,L_{l,n_l}\}\}$ to denote CNF. Any finite disjunction of literals is a *clause*. The set of atomic formulae whose literals belong to clause C and formula F are denoted by $\mathcal{V}(C)$ and $\mathcal{V}(F)$, respectively. A formula in CNF (finite set of clauses) is called a *clausal formula*. So, the above formula can be denoted as $F = \{C_1, ..., C_l\}$, where $Ci = \{L_{i,1}, ...,L_{i,n_i}\}$ (from now on, $l \geq 1$ is assumed). We denote the *empty clause,* the one without any literal, by $\square$. A clause with only one literal is called a *unit clause*.

To solve the incremental #SAT problem, we need some new notations. Let $C_1,..., C_s$ be clauses over V ($s \geq 1$). We denote by $m_V(C_1, ..., C_s)$ the number of *missing* variables from the union $C_1 \cup ... \cup C_s$, that is, $|\{A \mid A \in V - \mathcal{V}(C_1 \cup .. \cup C_s)\}|$. For example, if $V = \{p, q, r\}$, then $m_V(\{p\}, \{\bar{q}\}) = 1$ and $m_V(\{p,q\}, \{\bar{q},r\}) = 0$.

The notation $dif_V(C_1,..., C_s)$ points out if there is a *difference* in the clauses $C_1, ..., C_s$ regardless the presence of a literal and its negation. That is, if $(\exists i, j \in \{1,...,s\}, i \neq j$, such as $\exists L \in C_i$ and $\bar{L} \in C_j)$ or if $(\exists i \in \{1,...,s\}$, such as $C_i = \square)$, then $dif_V(C_1, ...,C_s) = 0$. Otherwise, $dif_V(C_1,...,C_s) = 2^{m_V(C_1,...,C_s)}$.

For example, if $V = \{p, q, r\}$, then $dif_V(\{p,q\}, \{\bar{q},r\}) = 0$ and $dif_V(\{p\}, \{\bar{q}\}) = 2$.

We denote by $det_V(C_1, ...,C_s)$ the number $2^{|V|} - \sum_{j=1}^{s} (-1)^{j+1} \sum_{1 \leq i_1 < ...<i_j \leq s} dif_V(C_{i_1}, ...,C_{i_j})$,

and we called it the **determinant** of the set of clauses $\{C_1, ..., C_s\}$. In fact, the determinant represents the number of truth assignments of the given clausal formula (Theorem 3.1). We called this number as *determinant* because of its similarities with the classical determinant from linear algebraic systems (both represent very useful numbers for those concepts). Because $C_1, ..., C_s$ can be permuted in any order, we may denote $det_V(C_1, ..., C_s)$ as $det_V(F)$, where $F = \{C_1, ..., C_s\}$. Next, useful properties of the determinant of a clausal formula are presented in Lemma 3.1 [And04]. They refer to the *monotony of the determinant* over the variables' alphabet and over the set of clauses. Thus, item a) refers to the monotony over V, item b) and c) refer to the empty and the unit clause, and items d), e), and f) refer to the inclusion rule and monotony over the set of clauses.

Lemma 3.1. *Let $F = \{C_1, ..., C_l\}$ be a clausal formula over V. Then:*

a) if $A_1, ..., A_m$ are atomic variables, $m \in \mathbb{N}_+$, $A_1, ..., A_m \notin V$, then $det_{V \cup \{A_1,...,A_m\}}(F) = 2^m.det_V(F)$;

b) if $\exists i \in \{1,..., l\}$, such as $C_i = \square$, then $det_V(F) = 0$;

c) if A is a new atomic variable, $A \notin V$, and $\{i_1, ..., i_s\}$ a subset of $\{1, ..., l\}$, $s \in \mathbb{N}_+$, then:

c1) $det_{V \cup \{A\}}(C_{i_1},..., C_{i_s}, \{A\}) = det_V(C_{i_1},..., C_{i_s})$

c2) $det_{V \cup \{A\}}(C_{i_1}, ..., C_{i_s}, \{\bar{A}\}) = det_V(C_{i_1}, ..., C_{i_s})$

d) if C_1 and C_2 are two clauses from F for which $C_1 \subseteq C_2$, then $det_V(F) = det_V(F - C_2)$.

e) let C_{l+1} and C_{l+2} be the two (new) clauses over V. If $C_{l+1} \subseteq C_{l+2}$ then $det_V(F \cup \{C_{l+1}\}) \leq det_V(F \cup \{C_{l+2}\})$;

f) let C_{l+1} be a new clause over V. Then $det_V(F \cup \{C_{l+1}\}) \leq det_V(F)$.

The next result makes the link between the determinant of a clausal formula and its satisfiability [And95, And04].

Theorem 3.1. *(Inverse Resolution Theorem) Let $F \in \mathbb{LP}$ over V. Then:*

(i) F is unsatisfiable $\Leftrightarrow$ $det_V(F) = 0$;

(ii) F is satisfiable $\Leftrightarrow det_V(F) \neq 0$. Much more, in this case there exist $det_V(F)$ number of truth assignments for F.

For a systematic computation of the determinant of a clausal formula $F = \{C_1, ..., C_l\}$ over V, it is better to use an *ordered labelled clausal tree*. The *full* clausal tree $CT(F) = (N, E)$, where N is the set of nodes and E is the set of edges, associated with F may be inductively constructed:

1)the zero (ground) level contains only a „dummy" root, that is an unlabelled node;

2)the first level contains, in order from the left to right the sequence of nodes labelled with: $(C_1, dif_V(C_1)),...,(C_l, dif_V(C_l))$.

3)for a given node v on the level k labelled with $(C_{i_k}, dif_V(C_{i_1},..., C_{i_k}))$, the level $k+1$ has the following direct descendants in this order, from the left to the right: $(C_{i_k+1}, dif_V(C_{i_1},..., C_{i_k}, C_{i_k+1}))$, ..., $(C_l, dif_V(C_{i_1},..., C_{i_k}, C_l))$.

The number of nodes of the full clausal tree $CT(F)$, without taking into account the „dummy" root, is the total number of elements of the sum which occur in $det_V(F)$. This number is exponential in l, *namely* $\binom{l}{1} + \binom{l}{2} + \binom{l}{3} + ... + \binom{l}{l} = 2^l - 1$. As a remark, $dif_V(C_{i_1},...,C_{i_s}) = 0$ implies $dif_V(C_{i_1},..., C_{i_s}, C_{i_{s+1}}) = 0$. It follows that only the nodes labelled with $(C_{i_k+1}, dif_V(C_{i_1}, ..., C_{i_k}C_{i_j}))$,for which $dif_V(C_{i_1}, ..., C_{i_k}, C_{i_j}) \neq 0$ and $j \in \{k+1, ..., l\}$, suffice to be generated for computing the determinant. The tree for which the nodes labelled with 0 are not generated is called the *ordered labelled reduced clausal tree,* and it is denoted as $CT_{red}(F) = (N_{red}, E_{red})$, where N_{red} and E_{red} are the set of nodes and edges, respectively. The reduced clausal tree has equal or fewer nodes that the full clausal tree.

The next example points out an ordered labelled reduced clausal tree attached to a particular clausal formula useful for computing the determinant.

Example 3.1. *Let $F = \{C_1, C_2, C_3, C_4, C_5\}$ be a clausal formula over $V = \{p, q, r, t\}$, where $C_1 = \{p, q\}$, $C_2 = \{p, r, t\}$, $C_3 = \{p, \bar{r}, \bar{t}\}$, $C_4 = \{q, r\}$, and $C_5 = \{\bar{p}, \bar{q}, \bar{r}\}$. Then $CT_{red}(F)$ is in figure 1.*

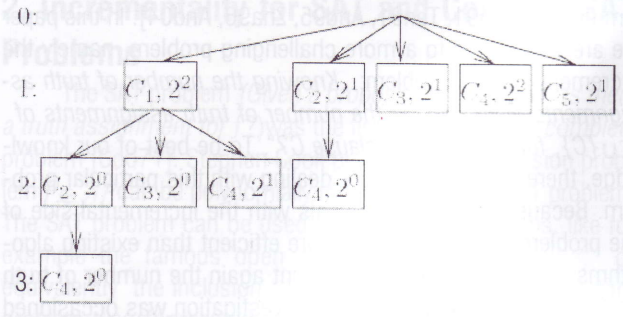

Figure 1. The ordered labeled reduced clausal tree

Adding the labels of the even levels and subtracting the labels of the odd ones, we obtain $det_V(F) = 2^4 - (2^2 + 2^1 + 2^1 + 2^2 + 2^1) + (2^0 + 2^0 + 2^1 + 2^0) - 2^0 = 6$. According to Theorem 3.1, F is satisfiable with 6 truth assignments. ▪

In [BiL99], it is mentioned that the algorithms for counting

truth assignments have something in common: the more variables have both negated and unnegated occurrences, the better is the performance of the algorithms on clausal formulae. This is approximately equivalent to say that CT_{red} will have much fewer nodes than CT (because many nodes in the full tree will have their dif_V labelled by 0). Computation of the determinant is faster for such cases.

4.Solving the Incremental Counting Satisfiability Problem

Since $CT_{red}(F)$ may have an exponential number of nodes depending on the number of clauses of F, whenever a new clause C is added, it is better to compute *only* the nodes which contain C and not the whole tree $CT_{red}(F \cup \{C\})$. But, the clausal tree $CT_{red}(F)$ attached to $F = \{C_1, ..., C_l\}$ cannot be used directly for incremental computing of $det_V(F)$ since the most recent clause (that is C_l) is spreaded as leaves of $CT_{red}(F)$, like in the left-hand side of *figure 2*.

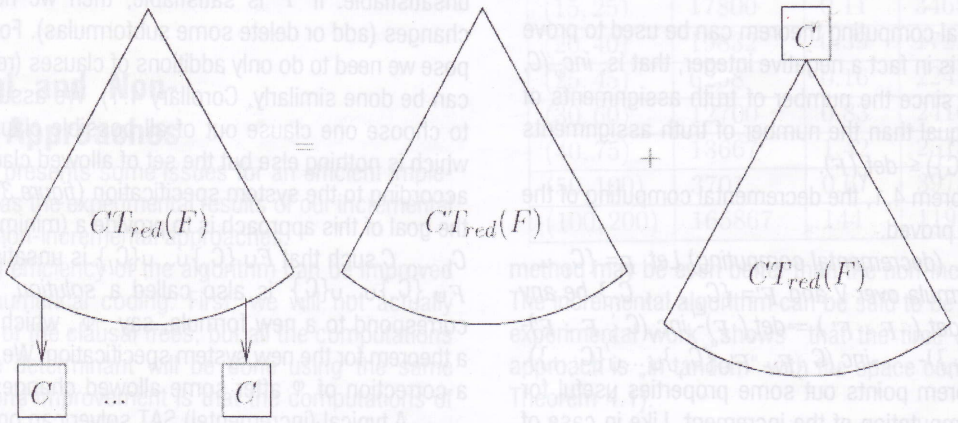

Figure 2. The incremental splitting of the clausal tree

To have a real incremental approach, we need a way to split the new clausal tree into the initial tree and a new tree containing the recent leaves. In other words, we need a procedure to move the nodes of $CT_{red}(F)$ such that C_l appears as a label only in the most recent clausal subtree, and to use the (old) value of $det_V(F)$. This procedure is highlighted in *Figure 2*, which shows that the new clausal tree from the left-hand side of *Figure 2* is splitted into the old clausal tree $CT_{red}(F)$ and a new clausal tree whose root is labelled with the clause C (both these trees are shown in the right-hand side of *figure 2*). In this way, we need to compute only the number of truth assignments of the new clausal tree. To do this counting, the notion of *increment* for a given clausal formula F and an arbitrary clause C is defined.

Notation 4.1. *If $F = \{C_1, ..., C_l\}$, where $l \geq 1$, is an arbitrary clausal formula over V and C is an arbitrary clause over V, then* $inc_V(C, F) = \sum_{s=0}^{l} (-1)^{s+1} \cdot \sum_{1 \leq i_1 < ... < i_s \leq l} dif_V(C, C_{i_1}, ..., C_{i_s})$ *is* called the increment of F with clause C.

As a remark for Notation 4.1, if $s = 0$, then the term from $inc_V(C, F)$ is $-dif_V(C)$. Similar like $det_V()$, the arguments of $inc_V()$, except the first one, can be permuted in any order. That is, given $(i_1, ..., i_l)$ an arbitrary permutation of $\{1, ..., l\}$, then $inc_V(C, C_{i_1}, ..., C_{i_l}) = inc_V(C, C_1, ..., C_l)$, where C is an arbitrary clause over V. Moreover, the increment of any clause C and any clausal formula $F = \{C_1, ..., C_l\}$ over V can be represented by *an ordered labelled clausal incremental tree*. The *full* clausal incremental tree $CIT(C, F) = (N, E)$ associated with C and F may be inductively constructed, where N and E denote the set of nodes and edges, respectively:

1) the first level contains the clause C as root, labelled with $(C, dif_V(C))$;

2) for a given node v on the level k, where $k \geq 1$, labelled with $(C_{ik}, dif_V(C, C_{i_1}, ..., C_{i_k}))$, the level $k+1$ has the following direct descendants in this order, from the left to the right: $(C_{i_k+1}, dif_V(C, C_{i_1}, ..., C_{i_k}, C_{i_k+1}))$, ..., $(C_l, dif_V(C, C_{i_1}, ..., C_{i_k}, C_l))$.

The number of nodes of the full clausal incremental tree $CIT(F)$, is the total number of elements of the sum which occur in $inc_V(C, F)$, that is $1 + \binom{l}{1} + \binom{l}{2} + \binom{l}{l} = 2^l$

Similar to $CT_{red}(F)$, the nodes whose *dif* are 0 need not be generated anymore. In other words, at step 2) of the above inductive construction, only the nodes labelled with $(C_{i_k+1}, dif_V(C, C_{i_1}, ..., C_{i_k}, C_{i_j}))$ are generated, where $j \in \{k+1, ..., l\}$ and $dif_V(C, C_{i_1}, ..., C_{i_k}, C_{i_j}) \neq 0$. We call this tree without these nodes the *ordered labelled reduced clausal incremental tree* associated with C and F and denote it as $CIT_{red}(C, F) = (N_{red}, E_{red})$.

Considering the formula F from Example 3.1, let us compute $inc_V(C_6, F)$, where $C_6 = \{\bar{p}, q\}$. Since $dif_V(C_6, C_1) = 0$, $dif_V(C_6, C_2) = 0$, $dif_V(C_6, C_3) = 0$, and $dif_V(C_6, C_5) = 0$, we get that $inc_V(C_6, F) = dif_V(C_6) + dif_V(C_6, C_4) = -2^2 + 2 = -2$. Theorem 4.1 allows us to say that, by adding C_6 to F, the number of truth assignments decreases with 2.

In the following, the main result is presented. It allows the computation of the determinant of a new clausal formula using the already computed determinant of the old clausal formula.

Theorem 4.1. *(incremental computing)Let* $F=\{C_1, ..., C_l\}$ *be a clausal formula over V and let* $F'=\{C_{l+1}, ..., C_{l+k}\}$, $k\geq 1$, *be a clausal formula over V. Then:*

a) the following identity holds:

(1) $det_V(F \cup F') = det_V(F) + inc_V(C_{l+1}, F) + inc_V(C_{l+2}, F \cup \{C_{l+1}\}) + ... + inc_V(C_{l+k}, F \cup \{C_{l+1}\} \cup ... \cup \{C_{l+k-1}\})$.

b) let us denote by N, N' the number of nodes of the reduced clausal trees corresponding to $det_V(F)$, $det_V(F \cup F')$, *respectively, and by* $N_{l+1}, N_{l+2}, ..., N_{l+k}$ *the number of nodes of the reduced clausal incremental trees corresponding to* $inc_V(C_{l+1}, F)$, $inc_V(C_{l+2}, F \cup \{C_{l+1}\})$, ..., *and* $inc_V(C_{l+k}, F \cup \{C_{l+1}\} \cup ... \cup \{C_{l+k-1}\})$, *respectively.Then the following identity holds:*

(2) $N' = N + N_{l+1} + N_{l+2} + ... + N_{l+k}$.

Item b) of incremental computing theorem says that the incremental computation of the determinant of a formula containing new clauses is *optimal*. That is, no new nodes are created in the new incremental clausal trees, except the ones which would have been created in the non-incremental approach. Subsection 6 of the Appendix contains a comparison of experimental results between the incremental and the non-incremental approaches.

The incremental computing theorem can be used to prove that the increment is in fact a negative integer, that is, $inc_V(C, F) \leq 0$. This holds since the number of truth assignments of $F \cup \{C\}$ is less or equal than the number of truth assignments of F, i.e. $det_V(F \cup \{C\}) \leq det_V(F)$.

Similar to Theorem 4.1, the decremental computing of the determinant can be proved.

Corollary 4.1. *(decremental computing) Let* $F=\{C_1, ..., C_l\}$ *be a clausal formula over V and* $F'=\{C_{i_1}, ..., C_{i_s}\}$ *be any subset of F. Then* $det_V(F - F') = det_V(F) - inc_V(C_{i_1}, F - F') - inc_V(C_{i_2}, F - F' \cup \{C_{i_1}\}) - ... - inc_V(C_{i_s}, F - F' \cup \{C_{i_1}\} \cup ... \cup \{C_{i_{s-1}}\})$.

The next theorem points out some properties useful for speeding up the computation of the increment. Like in case of the determinant, these properties refer to the *monotony of the increment* over the variables alphabet and over the clauses set. Thus, item a) refers to the monotony over *V*, items b) and c) refer to the case when the increment is zero, and items d), e), and f) refer to the inclusion rule and monotony over the set of clauses. For example, items b) and c) are important because the clauses of increments zero do not contribute to the value of the determinant, so they can be removed. Item f) proves that the increment of smaller clauses will be less than the increment of bigger clauses. Since an increment is a negative integer, it follows that is better to have small increments in order to get a new determinant close to zero.

Theorem 4.2. *Let* $F=\{C_1, ..., C_l\}$ *be a clausal formula over V. Then:*

a) If V' *is an alphabet such that* $V \subseteq V'$ *and C an arbitrary clause over V, then* $inc_{V'}(C, F) = 2^{|V'|-|V|} inc_V(C, F)$;

b) If $det_V(F) = 0$ *and C an arbitrary clause over V, then* $inc_V(C, F) = 0$;

c) if C_{l+1} and C_{l+2} are two (new) clauses over V and $inc_V(C_{l+1}, F) = 0$, then $inc_V(C_{l+2}, F) = inc_V(C_{l+2}, F \cup \{C_{l+1}\})$;

d) if A is an atomic variable, $A \notin V$, then $inc_{V \cup \{A\}}(\{A\}, F) = inc_{V \cup \{A\}}(\{\bar{A}\}, F) = -det_V(F)$;

e) If $C_1 \subseteq C_2$ *then* $inc_V(C_2, \{C_1, C_3, ..., C_l\}) = 0$ *and* $det_V(C_1, C_3, ..., C_l) = det_V(C_2, C_3, ..., C_l) + inc_V(C_1, \{C_2, C_3, ..., C_l\})$;

f) if C_{l+1} *and* C_{l+2} *are two (new) clauses over V and* $C_{l+1} \subseteq C_{l+2}$, *then* $inc_V(C_{l+1}, F) \leq inc_V(C_{l+2}, F)$.

5. Incremental Counting SAT Versus Incremental SAT

Since an (incremental) SAT solver finishes its execution after detecting the first truth assignment, it is expected that a (incremental) SAT solver to be faster than a (incremental) counting SAT solver. In contrast, an (incremental) counting SAT solver will count how many truth assignments exist for a given clausal formula. However, there exist large classes of problems where an (incremental) counting SAT solver outperforms a SAT solver. Re-design and debugging problems are such examples. A successful example is a redesign approach for timing constraints verification of real-time systems [AnC04].

As we mention in the Introduction, to prove that a formula ϕ is a theorem, a common technique is to show that its negation (that is, $\neg \phi$) is unsatisfiable. Most of the techniques reduce to show that a corresponding propositional formula F is unsatisfiable. If F is satisfiable, then we need to do some changes (add or delete some subformulas). For simplicity, suppose we need to do only additions of clauses (removal of clauses can be done similarly, Corollary 4.1). We assume that we have to choose one clause out of all possible clauses $C_{1,1}, ..., C_{1,k}$, which is nothing else but the set of allowed clauses to be added according to the system specification (*figure 3*). In other words, the goal of this approach is to provide a (minimal) set of clauses $C_1, ..., C_l$ such that $F \cup \{C_1\} \cup ... \cup \{C_l\}$ is unsatisfiable. Note that $F \cup \{C_1\} \cup ... \cup \{C_l\}$ is also called a *solution*. The solution will correspond to a new formula, say $\neg\varphi'$, which implies that φ' is a theorem for the new system specification. We may say that φ' is a correction of φ after some allowed changes.

A typical (incremental) SAT solver can only check whether $F \cup C_{1,1}, ..., F \cup C_{1,k}$ are satisfiable or not. But it cannot predict which of these *k* clauses lead to the solution. Instead, an (incremental) counting SAT solver is able to precisely decide which of the formulas $F \cup C_{1,1}, ..., F \cup C_{1,k}$ lead faster to a solution. This can be done efficiently by considering *j* from {1, ..., *k*} such that $inc_V(C_{1,j}, F)$ is the minimum among all increments $inc_V(C_{1,1}, F), ..., inc_V(C_{1,k}, F)$. In other words, the sub-tree of root $C_{1,j}$ will be selected for the next iterations (the one drawn with ticker lines, *Figure 3*). According to Theorem 4.1, we get that $C_{1,j}$ is the best candidate which minimize $det_V(F \cup C_{1,1}), ..., det_V(F \cup C_{1,k})$. - That is, $det_V(F \cup C_{1,j}) = det_V(F) + inc_V(C_{1,j}, F)$ is the minimum possible determinant. So, $C_{1,j}$ represents the optimal solution for C_1. The iteration can continue similarly with $C_2, ..., C_l$. This procedure continues until the determinant becomes zero, which corresponds to the unsatisfiability of the formula (Theorem 3.1). In conclusion, a SAT solver needs to visit an **exponential** number of nodes in the worst case (that is, the total number of nodes of the tree in *Figure 3*), while our #SAT solver needs to visit only a **linear** number of nodes in the worst case.

The above algorithm has a finite number of iterations according to monotonies of the determinant (Lemma 3.1) and the increment (Theorem 4.2). At any iteration, the determinant of F is decreasing because the increments of value zero are not taken into consideration. Moreover, the minimum increment

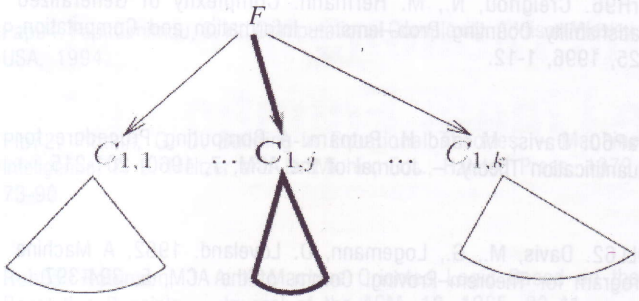

Figure 3. Counting SAT versus SAT

is chosen at any iteration, so based on the monotony, we may say that this algorithm is *optimal*.

Another advantage is the re-use of the old value of the determinant to the next iteration. The former computations are not repeated. As mentioned in Theorem 4.1, the number of nodes for computing the increment is optimal. Obviously, the incremental approach is much faster than the non-incremental approach. A comparison between these approaches has been presented the next section.

6.Incremental and Non-incremental Approaches

This section presents some issues for an efficient implementation, as well as the experimental results of our incremental approach against non-incremental approaches.

The practical efficiency of the algorithm can be improved by adopting the numerical coding. First, we will not actually „create" any node of the clausal trees, but all the computations needed to get the determinant will be done using the same memory. The second improvement is that the computations of powers of 2 can be avoided, by considering just its exponents.

The clausal formula $F = \{C_1, ..., C_l\}$ is said to be *uniformly random generated* with the probability $p = (p_1, p_2, 1- p_1- p_2)$ if in any clause C_i, any literal L appears positive or (exclusive) negative, with the probability p_1, respectively p_2, or does not appear in C_i with the probability $1- p_1- p_2$.

We have implemented the determinant and the increment computation algorithms. We did some experiments ran on a Pentium IV, 1.6GHz, and measure the time spend (in seconds) by the incremental computing of the determinant. For simplicity, we considered only the addition of two new clauses to the initial clausal formula $F = \{C_1, ..., C_l\}$ over the same set of variables $V = \{A_1, ..., A_n\}$. Moreover, we suppose that the probability of the literals in the clauses equals to $(\frac{1}{10}, \frac{1}{10}, \frac{8}{10})$. For short, we denote $CT_{red}(F \cup \{C_{l+1}\} \cup \{C_{l+2}\})$ by CT^{new}_{red}, $CIT_{red}(C_{l+1}, F)$ by CIT^1_{red}, and $CIT_{red}(C_{l+2}, F \cup \{C_{l+1}\})$ by CIT^2_{red}. Our testing instances refer to different values for (n, l).

For example, looking at the first lines of the tables, ($n = 10$ and $l = 20$), we may validate item b) of Theorem 4.1, namely 28831 = 12655+ 1760+ 14416. Moreover, the time needed for computing $det_v(F \cup \{C_{l+1}\} \cup \{C_{l+2}\})$ is approximately equal to the time consumed by the computation of $det_v(F \cup \{C_{l+1}\})$, $inc_v(C_{l+1}, F)$, and $inc_v(C_{l+2}, F \cup \{C_{l+1}\})$ altogether.

For the first line, we may see that 0.16≈0.06+ 0.01+ 0.05. One to memory caching, the time needed for the incremental method may be even better than the non-incremental method. The incremental algorithm can be said to be efficient since the experimental work „shows" that the time complexity of our approach is „in tandem" with the space complexity (item b of Theorem 4.1).

Table 1. The non-incremental approach

(n, l)	CT^{new}_{red}		$CT_{red}(F)$	
	Number of nodes	Time (sec.)	Number of nodes	Time (sec.)
(10, 20)	28831	0.16	12655	0.06
(15, 25)	70255	0.37	17799	0.13
(20, 40)	136714	3.32	99671	2.48
(25, 45)	78468	2.18	49800	1.50
(30, 60)	178531	7.70	141663	6.03
(40, 75)	150693	11.64	111837	8.77
(50, 100)	312276	39.26	268790	33.57
(100, 200)	2258144	2147	2080358	1992

Table 2. The incremental approach

(n, l)	CIT^1_{red}		CIT^2_{red}	
	Number of nodes	Time (sec.)	Number of nodes	Time (sec.)
(10, 20)	1760	0.01	14416	0.05
(15, 25)	17800	0.11	34656	0.21
(20, 40)	19832	0.39	17211	0.41
(25, 45)	6258	0.16	22410	0.71
(30, 60)	12700	0.83	24168	1.28
(40, 75)	13667	1.42	25189	2.19
(50, 100)	3701	0.67	39785	5.66
(100, 200)	165867	144	11919	30.48

7. Conclusion

We showed how our incremental counting SAT approach is a better alternative to an usual SAT solver when considering the challeging problems of re-design or debugging of systems. Our previous works [AnC04, ACCL05] have demonstrated that this approach is indeed a promising technique. Our #SAT solver can identify deterministically the solution, while a SAT solver needs to compute non-deterministically all the possible candidates for the solution (*figure 3*).

An interesting future work is to assign each new redesign choice occurrence a probability that may replace the designer guidance. This is especially useful when it is no possible to consult the designer for approval.

REFERENCES

AbM86. Abadi, M., Z. Manna. A Timely Resolution. Proc. First Annual Symposium on Logic in Computer Science,1986,176-186.

And95. Andrei, S. The Determinant of the Boolean Formulae. Scientific Annals of Bucharest University, Computer Science Section, Ano. XLIV, Romania, 1995, 83-92.

And04. Andrei, S. Inverse Propositional Resolution. – *Artificial Intelligence Review*, 22, 2004, No.4.

AnC04. Andrei, S., W.- N. Chin. Incremental Satisfiability Counting for Real-Time Systems. Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004, 482-489.

ACCL05. Andrei, S., W.-N. Chin, A. Cheng, and M. Lupu. Systematic Debugging of Real–time Systems Based on Incremental Satisfiability Counting. Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 05), 2005, 519528.

AFFHPV05. Armoni, R., L., Fix, R., Fraer, S., Huddleston, N., Piterman, and M. Y. Vardi. SAT-based Induction for Temporal Safety Properties. Electronic Notes in Theoretical Computer Science, 2005.

BDBGS04. Barrett, C., D.L.,Dill, S.,Berezin, V., Ganesh and A. Stump. Cooperating Validity Checker (CVC), 2004, http://verify.stanford.edu/CVC/.

BiL99. Birbaum, E. and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. – *Journal of Artificial Intelligence Research*, 10, 1999, 457-477.

Ble77. Bledsoe, W. W. Non-resolution Theorem Proving. – *Artificial Intelligence*, 9, 1977, 1-35.

BMS75. Boyer, R. S. and J. S. Moore. Proving Theorems about LISP Functions. – *J. ACM*, 22, No.I, 1975, 83-105.

Bry86. Bryant, R. E. Graph-based Algorithms for Boolean Function Manipulation. – IEEE Transactions on Computers, 35, no.8, 1986, 677-691.

Bry92. Bryant, R. E. Symbolic Boolean Manipulation with Ordered Binary-decision Dia–grams. – *ACM Computing Surveys*, 40, No.2, 1992, 293-318.

ChL73. Chang, C. L. and R. C. T. Lee, Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York, 1973.

Cla2000. Clay Mathematics Institute, 2000. http://www.claymath.org/MillenniumPrizeProblems/PvsNP/.

Coo71. Cook, S. A. The Complexity of Theorem-proving Procedures. Proc. Third Annual ACM Symposium Theory of Computing, 1971, 151-158.

CFFGKTV01. Copty, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y.Vardi. Benefits of Bounded Model Checking at an Industrial Setting. Proceedings of the 13th International Conference on Computer Aided Verification, 2102 of LNCS, Springer Verlag, 2001, 436-453.

CrH96. Creignou, N., M. Hermann. Complexity of Generalized Satisfiability Counting Prob–lems. – Information and Computation, 125, 1996, 1-12.

DaP60. Davis, M. and H. Putnam. A Computing Procedure for Quantification Theory. – Journal of the ACM, 7, 1960, 201-215.

DLL62. Davis, M., G., Logemann, D. Loveland. 1962, A Machine Program for Theorem–Proving. Comm. of the ACM, 5, 394-397.

Dub91. Dubois, O. Counting the Number of Solutions for Instances of Satisfiability. Theo–retical Computer Science, 81, 1991, 49-64.

GaJ79. Garey, M. R. and D. S.Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness. Freeman, San Francisco, CA, 1979.

GGST00. Giunchiglia, E., F. Giunchiglia, R. Sebastiani, T. Tacchella. SAT Vs. Translation Based Decision Procedures for Modal Logics: a comparative evaluation. – Journal of Applied Non-Classical Logics, 2000, No.2,10.

Hor98. Horrocks, I. Using an Expressive Description Logic: FaCT or Fiction? Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98), A. Cohn, L. Schubert, and S. Shapiro, Eds., Morgan Kaufmann, 636-647.

Hoo93. Hooker, J. N. Solving the Incremental Satisfiability Problem. – J. Logic Programming, 15, 1993, 177-186. ics05. ***: 2005, Integrated Canonizer and Solver, http://fm.csl.sri.com/.

Iwa89. Iwana, K. CNF Satisfiability Test by Counting and Polynomial Average Time. – Siam J. Comput., 18, No.2, 1989, 385-391.

JaM86. Jahanian, F. and A. Mok. Safety Analysis of Timing Properties in Real-Time Systems. – IEEE Transactions on Software Engineering, SE-12, 1986, No.9, 890-904.

JaM87. Jahanian, F. and A. Mok. A Graph-Theoretic Approach for Timing Analysis and its Implementation. – *IEEE Transactions on Computers*, C-36, 1987, No.8, 961-975.

Lev73. Levin, L. A. (Submitted: 1972, Reported in Talks: 1971) Universal Search Problems. *Problemy Peredachi Informatsii = Problems of Information Transmission*, 9, 1973, no. 3, 265-266.

McM93. McMillan, K. L. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publisher, 1993.

MMZZM2001. Moskewicz, M. W. C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: En–gineering an Efficient Sat Solver. In DAC '01: Proceedings of the 38th Conference on Design Automation, ACM Press, 2001, 530-535.

Pap94. Papadimitriou, C. H.: Computational Complexity. Addison Wesley, USA, 1994.

Plo72. Plotkin, G. D. Building-in Equational Theories. – Machine Inteligence, 7, B. Meltzer and D. Michie, Eds., Halsted Press, 1972, 73-90.

Rob65. Robinson, J. A. A Machine Oriented Logic Based on the Resolution Principle. – Journal of the ACM, 12, 1965, 23-41.

Rot96. Roth, D. On the Hardness of Approximate Reasoning. – Artificial Intelligence, 82, 1996, 273-302.

Sim75. Simon, J. On Some Central Problems in Computational Complexity. Doctoral Thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, 1975.

smv ***: SMV, http://nusmv.irst.itc.it/.

smvsat02.***:2002,SMVSAT, http://www.mimuw.edu.pl/sl/teaching/PMW/SMV-doc/releasenotes.html.

Sto77. Stockmeyer, L. J. The Polynomial-time Hierarchy. – Theoretical Computer Science, 3, 1977,1-22.

Tan91. Tanaka, Y. A Dual Algorithm for the Satisfiability Problem. – Information Processing Letters, 37,1991, 85-89.

Tra84. Trakhtenbrot, B. A. A Survey of Russian Approaches to Perebor (Brute-force Search) algorithms. – Annals of the History of Computing, 6,1984, No.4, 384-400.

Val79. Valiant, L. G. The Complexity of Enumeration and Reliability Problems. – SIAM Journal on Computing, 8, 1979, 410-421.

WKS01. Whittemore, J., J., Kim, K.Sakallah. SATIRE: A New Incremental Satisfiability Engine. Proceedings of the 38th ACM/IEEE Conference on Design Automation, 2001, 542-545.

Zha96. Zhang, W.:1996, Number of Models and Satisfiability of Sets of Clauses.–Theoretical Com–puter Science,155,1996, 277-288.

Manuscript received on 19.11.2007

Contacts:

Stefan Andrei
Lamar University, Department of Computer Science, Texas, USA,
e-mail: sandrei@cs.lamar.edu

Albert M. K. Cheng
University of Houston, Department of Computer Science, Texas, USA,
e-mail: cheng@cs.uh.edu

Gheorghe Grigoras
Cuza University of Iasi, Department of Computer Science, Romania,
e-mail: grigoras@infoiasi.ro

Lawrence J. Osborne
Lamar University, Department of Computer Science, Texas, USA,
e-mail: osborne@cs.lamar.edu