Incremental Verification of Timing Constraints for Real-Time Systems

Ştefan ANDREI [1], Wei-Ngan CHIN [1], and Martin RINARD [2]
[1] National University of Singapore, [2] Massachusetts Institute of Technology
Singapore-MIT Alliance E4-04-10, 4 Engineering Drive 3, Singapore 117576

***Abstract*— Testing constraints for real-time systems are usually verified through the satisfiability of propositional formulae. In this paper, we propose an alternative where the verification of timing constraints can be done by counting the number of truth assignments instead of boolean satisfiability. This number can also tell us how "far away" is a given specification from satisfying its safety assertion. Furthermore, specifications and safety assertions are often modified in an incremental fashion, where problematic bugs are fixed one at a time. To support this development, we propose an incremental algorithm for counting satisfiability. Our proposed incremental algorithm is *optimal* as no unnecessary nodes are created during each counting. This works for the class of *path* RTL ([1], [5]). To illustrate this application, we show how incremental satisfiability counting can be applied to a well-known rail-road crossing example, particularly when its specification is still being refined.**

***Index Terms*— Real-time infrastructure and development, timing constraint, #SAT problem, incremental computation**

I. Introduction

Real-time systems can be defined either by a structural specification (how its components work) or by a behavioral specification (showing the response of each component in response of an internal or external event). A behavioral specification often suffices for verifying the timing properties of the system. Given the behavioral specification of a system (denoted by SP) and a safety assertion (denoted by SA) to be analysed, the goal is to relate a given safety assertion with the system specification [1]. If SA is a theorem derivable from SP, then the system is *safe*. If SA is unsatisfiable, then the system is inherently unsafe. If $\neg SA$ is satisfiable under certain conditions, additional constraints may be added to ensure its safety. Our work is targetted to this scenario where we introduce an incremental approach to obtain a modified safety assertion as theorem, as outlined in **Algorithm A** below.

Input: SP, SA such that $\neg SA$ is satisfiable;
Output: SP_{new}, SA_{new} such that the system is safe;
Method:

1. $k = 1$; $SP_1 = SP$; $SA_1 = SA$;
2. `while (`$SP_k \rightarrow SA_k$ is not a tautology`) {`
3. let SP_{new} and SA_{new} be new constraints;
4. $SP_{k+1} = SP_k \oplus SP_{new}$;
5. $SA_{k+1} = SA_k \oplus SA_{new}$;
6. $k = k + 1$; `}`
7. $SP_{new} = SP_k$; $SA_{new} = SA_k$;

The satisfiability of the formula $SP_{k+1} \rightarrow SA_{k+1}$ can be expressed incrementally from the satisfiability of $SP_k \rightarrow SA_k$. The total cost of the new method can be more efficiently achieved through computing the satisfiability of the newly added or subtracted clauses, according to the operator $\oplus$, when compared to the old formula, and not from the satisfiability of the entire new formula. Our method requires the debugging of the real-time system at step **3**. We correlate this with the incremental computation for the satisfiability of $SP_k \rightarrow SA_k$. Our approach does not require us to re-compile the whole system, as we could incorporate the new constraints by re-using most of the older formula. In general, automatic debugging is hard. To assist in this direction, we will provide a systematic way of debugging with the help of incremental counting satisfiability. We illustrate this with the well-known railroad crossing example, used in [2], [3], as case study.

Real-time logic (RTL), which is based on a first-order logic with restricted features, was introduced in [4] to capture the timing requirements of real-time systems. The problem of proving the safety assertion from its specification is in general undecidable for the full set of RTL formulas based on the Presburger Arithmetic. The correctness of a real-time system can be achieved by computing the satisfiability of an associated propositional formula. We shall consider an RTL class of formulas (invented in [1]), with the following restrictions:

a) each arithmetic inequalities may involve only two terms and an integer constant, where a term is either a variable or a function and

b) no arithmetic expressions that have a function taking an instance of itself as an argument.

This subclass of RTL formulas (also called *path* RTL, [5]) exploits an efficient constraint-graph technique in integer programming [1]. Despite these restrictions, this constraint graph technique (also called *refutation by positive cycles*) is still undecidable [5]. Moreover, in [5], it is proved that the refutation by positive cycles is incomplete for path RTL (that is, even if the constraints graph attached to the formula has no cycles, it may happen that the formula is still unsatisfiable).

This paper represents an updated and corrected version of the paper *Incremental Satisfiability Counting for Real-Time Systems. IEEE Real-Time and Embedded Technology and Applications Symposium*, 25-28 of May, 2004

Ştefan ANDREI is with Singapore-MIT Alliance, National University of Singapore, CS Programme, Singapore, 117543; e-mail: andrei@comp.nus.edu.sg

Wei-Ngan CHIN is with National University of Singapore, School of Computing, Department of Computer Science, Singapore, 117543; e-mail: chinwn@comp.nus.edu.sg

Martin RINARD is with MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139; e-mail: rinard@lcs.mit.edu.

Despite this, Wang and Mok mentioned that the refutation by positive cycles method is believed to be a natural technique for reasoning about timing inequalities. Furthermore, they presented a polynomial time algorithm for the positive cycle detection. To get the decidability and completeness of the path RTL, more restrictions have to be added, so that is the case of *semi-periodical* RTL [5]. Informally, this subclass requires that the occurrence of every event type exhibits a periodical behavior and has infinitely many occurrences. The advantages of this subclass are that the satisfiability problem is decidable and the positive cycle detection is complete for the problem. In our paper, we consider the refutation by positive cycle for the path RTL. Of course, if the real-time system exhibits a periodical behavior for every event occurrence, then our technique may benefit from the completeness of the positive cycle detection technique.

The class of path RTL formulas is very practical and expressive ([1], [3]). For example, it was used to describe the timing properties of a moveable control rods in a reactor [1] and of the X-38, an autonomous spacecraft designed and built by NASA as a prototype of the International Space Station Crew Return Vehicle [6].

Section II presents a motivating example in path RTL. Section III presents preliminary results regarding #SAT problem. Section IV developed our approach for computing the value of the determinant in an incremental way. This section contains the main results (Theorem 4.1, Corollaries 4.1 and 4.2), Algorithm **B** together with some practical improvements. Subsection V solves the railroad crossing problem through an incremental approach. Take note that the debugging of the specification can be achieved by analysing the constraint graph. The last two sections present related work and conclusions.

II. Motivating Example in RTL

Real-Time Logic provides a uniform way for the specification of relative and uniform timing of events. It is an extension of integer arithmetic without multiplication (Presburger arithmetic) that adds a single uninterpreted binary *occurrence function*, denoted by @, to represent the relationship between events of a system, and their times of occurrence. The equation $@(e,\ i) = t$ states that the time of the $i-$th occurrence of event e is t. Let us denote $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_+$ the set of integers, positive integers, and strict positive integers, respectively. The time occurrence function is a mapping @: $E \times \mathbb{N}_+ \rightarrow \mathbb{N}$, where E is a domain of events, and such that @ is strictly monotonically increasing in its second argument, i.e. $@(E,\ i) < @(E, i+ 1)$, for any $i \in \mathbb{N}_+$. There are no event variables, or uninterpreted predicate symbols. So, RTL formulas are boolean combinations of equality and inequality predicates of standard integer arithmetic, where the arguments of the relations are integer valued expressions involving variables, constants, and applications of the function symbol @. Usually, there are four classes of events, namely: stop and start events ($\uparrow A$ and $\downarrow A$ denote the start and stop events of the action A), transition events and external events (prefixed with Ω).

To illustrate this, consider the railroad crossing example. Its behavioral specification (denoted as SP) is described in natural language [3] as follows:

"*When the train approaches the sensor, a signal will initiate the lowering of the gate",* **and** *"Gate is moved to the down position within 30s from being detected by the sensor",* **and** *"The gate needs at least 15s to lower itself to the down position*".

The goal of this real-time system is described by the following safety assertions (denoted as SA):

"**If** *the train needs at least 45s to travel from the sensor to the railroad crossing",* **and** *"the train crossing is completed within 60s from being detected by the sensor",* **then** *"we are assured that at the start of the train crossing, the gate has moved down* **and** *that the train leaves the railroad crossing within 45s from the time the gate has completed moving down*".

Let $\mathbb{LP}$ be the *propositional logic* over the finite set of *atomic formulae* (known also as *propositional variables*) denoted by $V = \{A_1, A_2, ..., A_n\}$. A *literal* is an atomic formula (*positive* literal), and so is its negation (*negative* literal). For any literal L, we put $\overline{L} = \neg A$ if $L = A$ and $\overline{L} = A$ if $L = \neg A$. If A is the atomic formula corresponding to L or $\overline{L}$, then we denote $\mathcal{V}(L) = \mathcal{V}(\overline{L}) = A$.

Any function $\mathcal{S} : V \rightarrow \{0, 1\}$ is a *structure* (known also as assignment, substitution, instance, and model) and it can be uniquely extended in $\mathbb{LP}$ to F (this extension will be denoted also by $\mathcal{S}$). The binary vector $(y_1, ..., y_n)$ is a truth assignment for F over $V = \{A_1, ..., A_n\}$ iff $\mathcal{S}(F) = 1$ such that $\mathcal{S}(A_i) = y_i,\ \forall\ i \in \{1, ..., n\}$. The formula $F|_{[y_i/A_i]}$ denotes F for which all the occurrences of variable A_i are replaced by y_i. If $F_1,\ F_2 \in \mathbb{LP}$ then $F_1 \equiv F_2$ (F_1 is *strongly equivalent* with F_2) if $\mathcal{S}(F_1) = \mathcal{S}(F_2)$ for any structure $\mathcal{S}$. We say that F_1 is *weakly equivalent* with F_2 ($F_1 \equiv_w F_2$) iff there exists a structure $\mathcal{S}$ such as $\mathcal{S}(F_1) = \mathcal{S}(F_2)$. A formula F is called *tautology* iff for any structure $\mathcal{S}$, it follows that $\mathcal{S}(F) = 1$. A formula F is called *satisfiable* iff there exists a structure $\mathcal{S}$ for which $\mathcal{S}(F) = 1$. 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): $F = (L_{1,1} \vee\ ... \vee L_{1,n_1}) \wedge\ ... \wedge (L_{l,1} \vee\ ... \vee L_{l,n_l})$, where $L_{i,j}$ are literals. In this paper, we shall use a set representation $F = \{\{L_{1,1},\ ...,\ L_{1,n_1}\},\ ...,\ \{L_{l,1},\ ...,\ L_{l,n_l}\}\}$ 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 $C_i = \{L_{i,1}, ..., L_{i,n_i}\}$ (from now on, $l \geq 1$ is assumed). In this paper, only *non-tautological* clauses (which have no simultaneous occurrences of a literal L and $\overline{L}$) are considered. We say that a clause C_1 is included in the clause C_2 (denoted by $C_1 \subseteq C_2$) iff $\forall\ L \in C_1$ we have $L \in C_2$. A finite non-tautological clause C constructed over V is *maximal* iff $\mathcal{V}(C) = V$. A clausal formula is *maximal* iff it contains only maximal clauses. We denote the *empty clause*, the one without any literal, by $\square$. A clause with only one literal is called a *unit clause*. A clause C is called *positive* (or *negative*) iff C contains only positive (or negative) literals.

Coming back to the problem of railroad crossing, we can express it in terms of path RTL, as follows:

$SP : \forall x$ ($@(TrainApproach,\, x) \leq @(\uparrow DownGate,\, x) \wedge @(\downarrow DownGate,\, x) \leq @(TrainApproach,\, x)+ 30) \wedge \forall y$ ($@(\uparrow DownGate,\, y)+ 15 \leq @(\downarrow DownGate,\, y)$)

$SA : \forall t\ \forall u$ ($@(TrainApproach,\, t)+ 45 \leq @(\uparrow TrainCrossing,\, u) \wedge @(\downarrow TrainCrossing,\, u) < @(TrainApproach, t)+ 60 \rightarrow @(\uparrow TrainCrossing, u) \geq @(\downarrow DownGate, t) \wedge @(\downarrow TrainCrossing, u) \leq @(\downarrow DownGate, t) + 45$)

In order to translate into an equivalent Presburger arithmetic formula, each $@(E, i)$ is replaced by a function $f_E(i)$. For example, $@(TrainApproach, x)$ will be $f(x)$, $@(\uparrow DownGate, x)$ will be $g_1(x)$, $@(\downarrow DownGate, x)$ will be $g_2(x)$, $@(\uparrow TrainCrossing, u)$ will be $h_1(u)$, $@(\downarrow TrainCrossing, u)$ will be $h_2(u)$, etc. So, the complete translation into the Presburger arithmetic formula becomes:

$SP : \forall x$ ($f(x) \leq g_1(x) \wedge g_2(x) \leq f(x)+ 30) \wedge \forall y$ ($g_1(y)+ 15 \leq g_2(y)$)

$SA : \forall t\ \forall u$ ($f(t)+ 45 \leq h_1(u) \wedge h_2(u) < f(t)+ 60 \rightarrow g_2(t) \leq h_1(u) \wedge h_2(u) \leq g_2(t) + 45$)

To show that $SP \rightarrow SA$ is a tautology is equivalent to proving that $SP \wedge \neg SA$ is unsatisfiable. The corresponding formula for $SP \wedge \neg SA$ can be translated into CNF and denoted by F_1, where every literal has the general form: $v_1 \pm I \leq v_2$, where v_1, v_2 are function occurrences and $I \in \mathbb{N}$ a constant. Here is the equivalent CNF form after skolemising ($[T/t][U/u]$ correspond to the $\neg SA$ part):

$SP : \forall x\ \forall y$ ($f(x) \leq g_1(x) \wedge g_2(x)- 30 \leq f(x) \wedge g_1(y)+ 15 \leq g_2(y)$)

$\neg SA : f(T)+ 45 \leq h_1(U) \wedge h_2(U)- 59 \leq f(t) \wedge (h_1(U)+ 1 \leq g_2(T) \vee g_2(T)+ 46 \leq h_2(U)$)

Next, the constraint graph is constructed (Figure 1). For each literal $v_1 \pm I \leq v_2$, two nodes labelled with v_1 and v_2 are linked by an edge (v_1, v_2) with weight $\pm I$. Thus, a set of inequalities represented by such a graph is unsatisfiable iff a cycle is present in the graph with a positive total weight on it [1]. It is straightforward to show that if all edges involved in a positive cycle in the constraints graph correspond to literals (inequalities) which belong to unit clauses, then F_1 must be unsatisfiable. However, if an edge in the cycle corresponds to a literal that belongs to a non-unit clause, then it is necessary to show that each of the remaining literals in this clause corresponds to an edge in a different positive cycle.

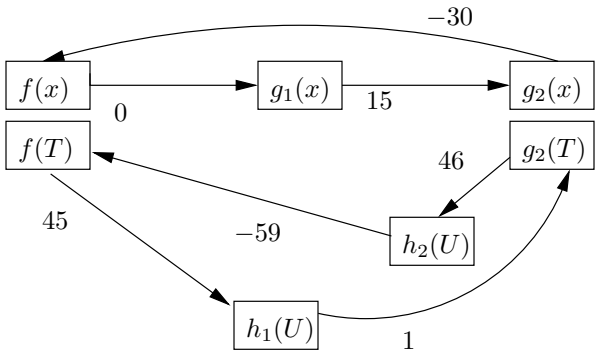

Figure 1. Railroad crossing constraint graph (1)

A variation of Herbrand's Theorem for this approach was presented in [1]. It says that: "*a set S of clauses is unsatisfiable iff there is a finite unsatisfiable set of ground instances of S and $\neg P_i\ \forall\ i\ \in\ \overline{1,n}$, where each P_i is the conjunction of inequalities corresponding to the edges in a positive cycle detected in the constraint graph for S*". The above formulation permits one to use any method in propositional logic to check for unsatisfiability as positive cycles are detected and the appropiate clauses are added to the existing set of clauses. Therefore, F_1 is satisfiable iff $F_1 \wedge \{\neg P_i \mid$ for all positive cycle $i\}$ is satisfiable.

The clausal formula F_1 contains only positive clauses corresponding to all edges of the constraint graph, and only negative clauses corresponding to a positive cycle. Even if each clause is positive or negative, the CNF satisfiability is NP-complete [1]. We make the following notations for the literals: $A_1 = f(x) \leq g_1(x)$, $A_2 = g_2(x)- 30 \leq f(x)$, $A_3 = g_1(y)+ 15 \leq g_2(y)$, $A_4 = f(T)+ 45 \leq h_1(U)$, $A_5 = h_2(U)- 59 \leq f(T)$, $A_6 = h_1(U)+ 1 \leq g_2(T)$, $A_7 = g_2(T)+ 46 \leq h_2(U)$. Therefore, F_1 has the positive clauses: $\{A_1\}$, $\{A_2\}$, $\{A_3\}$, $\{A_4\}$, $\{A_5\}$, $\{A_6, A_7\}$.

To use the positive cycles, we have two methods: a *dynamic* one and a *global* one. The dynamic algorithm [1] means that as each new cycle is detected, the corresponding clause is added to the set of existing ones, and is then checked for unsatisfiability. It if is shown to be unsatisfiable, we can stop immediately. Otherwise, it is necessary to continue the node removal operation until another positive cycle is found. An equivalent approach says that we may identify all the positive cycles and add all of them to F_1 from the beginning (no node removal is required).

Using the second above approach, three positive cycles in the constraint graph have been identified (Figure 1), so F_1 has the negative clauses: $\{\overline{A_2}, \overline{A_4}, \overline{A_6}\}$, $\{\overline{A_4}, \overline{A_5}, \overline{A_6}, \overline{A_7}\}$, $\{\overline{A_1}, \overline{A_3}, \overline{A_5}, \overline{A_7}\}$. Of course, the unification of the first-order logic is applied, e.g. the nodes labelled with $f(x)$ and $f(T)$ are considered as one using the substitution $[T/x]$ ([1], [3]). At the end of Section III, we shall see that F_1 is unsatisfiable, so $SP \wedge \neg SA$ is too. Thus, $SP \rightarrow SA$ is a theorem, i.e. the real-time system is safe.

We propose to embed the incremental computation of the determinant of a clausal formula in the verification of timing constraints of a real-time system. This will tell us how "far away" is the current specification from satisfying the safety assertion. The modification of the specification and/or safety assertions is useful for incremental debugging, in which bugs in problematic areas are fixed one at a time until the system is safe [7]. We choose this approach in order to benefit from incremental debugging, which includes not re-computing the satisfiability of the whole problem every time.

III. Preliminary Results

The basic incremental satisfiability problem of propositional logic has been introduced in [8], as follows: "*Given a propositional formula F, check whether $F \cup \{C\}$ is satisfiable for a given clause C*". The algorithm presented in [8] solves the SAT problem using the Davis-Logemann-Loveland's procedure [9] combined with a backtracking strategy that adds one clause at a time. In [10], a SAT solver able to handle non-

conjunctive normal form constraints and incremental satisfiability was presented. For efficiency reasons, our technique is applied incrementally. The incremental #SAT problem says that "*Knowing the number of truth assignments of F, what is the number of truth assignments of $F \cup \{C\}$, for any arbitrary clause C*".

In this section we fix some concepts and notations [11] to allow the text to be self contained, by including some results and examples. 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 $\frac{n!}{(n-i)!\cdot i!}$, where $n! = 2\cdot 3\cdot \ ... \cdot n$.

Notation 3.1: Let $C_1, ..., C_s$ be clauses over V ($s \geq 1$). We denote:

a) $m_V(C_1, ..., C_s) = |\{A \mid A \in V - \mathcal{V}(C_1 \cup ... \cup C_s)\}|$;

b) $dif_V(C_1, ..., C_s) =$

b1) 0 if ($\exists\ i, j \in \{1, ..., s\}$, $i \neq j$, such as $\exists\ L \in C_i$ and $\overline{L} \in C_j$) **or** if ($\exists\ i \in \{1, ..., s\}$, such as $C_i = \square$);

b2) $2^{m_V(C_1, ..., C_s)}$ otherwise;

c) $det_V(C_1, ..., C_s) = 2^{|V|} - \sum_{j=1}^{s} (-1)^{j+1} \cdot \sum_{1 \leq i_1 < ... < i_j \leq s} dif_V(C_{i_1}, ..., C_{i_j})$ is called the **determinant** of the set of clauses $\{C_1, ..., C_s\}$. ∎

Because the arguments of $det_V()$ can be permuted in any order, we may denote $det_V(F) = det_V(C_1, ..., C_l)$, where $F = \{C_1, ..., C_l\}$. Next, some useful properties of the determinant of a clausal formula will be presented. We show how the determinant of a clausal formula will be affected if we consider some particular forms of clauses/rules such as: the empty clause, the unnecessary variables rule, the inclusion rule. Lemma 3.1 will be intensively used in Algorithm **B** described in the next section.

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

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

b) 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:

b1) $det_{V \cup \{A\}}(C_{i_1}, ..., C_{i_s}, \{A\}) = det_V(C_{i_1}, ..., C_{i_s})$.

b2) $det_{V \cup \{A\}}(C_{i_1}, ..., C_{i_s}, \{\overline{A}\}) = det_V(C_{i_1}, ..., C_{i_s})$.

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

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

The next result makes the link between the determinant of a clausal formula and its satisfiability [11]. An equivalent result, but proved differently, has also been presented in [12].

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

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

(ii) F is satisfiable $\iff det_V(F) \neq 0$. 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 $CT(F) = (N, E)$ 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 $\binom{l}{1} + \binom{l}{2} + \binom{l}{l} = 2^l - 1$.

Remark 3.1: Since $dif_V(C_{i_1}, ..., C_{i_s}) = 0$ implies $dif_V(C_{i_1}, ..., C_{i_s}, C_{i_{s+1}}) = 0$, then only the nodes labelled with $(C_{i_k+1}, dif_V(C_{i_1}, ..., C_{i_k}, C_{i_j}))$, where $dif_V(C_{i_1}, ..., C_{i_k}, C_{i_j}) \neq 0$ and $j \in \{k+1, ..., l\}$, are enough 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 $CT_{red}(F) = (N_{red}, E_{red})$. 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 $C_1 = \{p, q\}$, $C_2 = \{p, r, t\}$, $C_3 = \{p, \overline{r}, \overline{t}\}$, $C_4 = \{q, r\}$, and $C_5 = \{\overline{p}, \overline{q}, \overline{r}\}$. Then $CT_{red}(F)$ is in Figure 2:

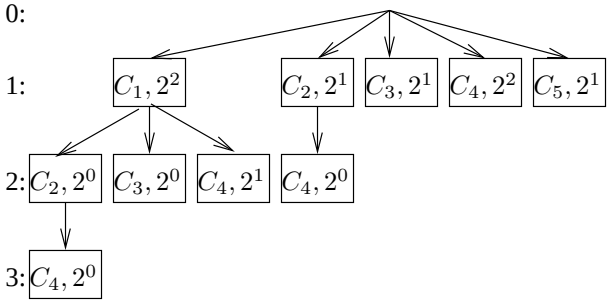

Figure 2. The ordered labeled reduced clausal tree

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 [13], 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). So, the computation of the determinant will be faster in this case.

Reconsidering the problem of railroad crossing described in Section I, we get that $det_{V_1}(F_1) = 0$, and 91 nodes are generated for $CT_{red}(F_1)$, where $V_1 = \{A_1, ..., A_7\}$.

IV. Incremental Computing of the Determinant

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 a leaf in all clausal sub-trees of $CT_{red}(F)$. Therefore, 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 sub-tree, and to use the (old) value of $det_V(F)$. Next, the increment of a given clausal formula F with an arbitrary clause C is defined.

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

It returns an integer number representing the number of truth assignments which have to be added or substracted from the previous value of the determinant. Similar to the determinant, 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:

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 $k \ge 1$, labelled with $(C_{i_k}, dif_V(C, C_{i_1}, ..., C_{i_k}))$, 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 $1 + \binom{l}{1} + \binom{l}{2} + \binom{l}{l} = 2^l$.

Again, 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 $CIT_{red}(C, F) = (N_{red}, E_{red})$.

Before presenting the main result of this section, a result which allows the permutation of the arguments of dif_V, det_V and inc_V is necessary.

Lemma 4.1: (permutation lemma) Let $F = \{C_1, ..., C_l\}$, be a clausal formula over V and $(i_1, ..., i_l)$ an arbitrary permutation of $\{1, ..., l\}$. Then $dif_V(C_{i_1}, ..., C_{i_l}) = dif_V(C_1, ..., C_l)$, $det_V(C_{i_1}, ..., C_{i_l}) = det_V(C_1, ..., C_l)$ and $inc_V(C, C_{i_1}, ..., C_{i_l}) = inc_V(C, C_1, ..., C_l)$, where C is an arbitrary clause over V.

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 $F' = \{C_{l+1}, ..., C_{l+k}\}$, $k \ge 1$, 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) $N' = N + N_{l+1} + N_{l+2} + ... + N_{l+k}$.

Because of the efficiency reasons, the incremental computing theorem is better to be applied only if the clauses from F' are new, that is $F' \cap F = \emptyset$.

Similarly 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 addition of a new clause C to a given clausal formula F over the same set of variables V will decrease the number of true assignments, i.e. $inc_V(C, F) \le 0$. The next corollar points out some situations when the computation of the increment can be speed up.

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

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

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

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

d) If $det_V(F) = 0$ and C an arbitrary clause over V, then $inc_V(C, F) = 0$.

In the following, we put together all the previous results, by providing Algorithm **B** which is able to compute the value of the determinant in an incremental manner.

Algorithm B

Input: $F = \{C_1, ..., C_l\}$ a clausal formula (over V);
Output: $det_V(F)$ computed in an incremental way and "No/Yes" corresponding to (un)/satisfiability of F.
Method:

```
main() {
1. det = 1; F_old = ∅; V_old = ∅;
2. for (int i = 1; i <= l; i++) {
3.   if (C_i == □) {
4.     det = 0; printf("No, F is unsatisfiable."); Exit}
5.   F_new = F_old ∪ {C_i}; V_new = V_old ∪ V(C_i);
6.   if (V_new != V_old) det = det * 2^|V(C_i) − V_old|;
7.   det = det − dif_{V_new}(C_i);
8.   if (i > 1 && C_j ⊈ C_i, ∀ j ∈ {1, ..., i − 1})
       inc(i,[C_i],2);
9.   F_old = F_new; V_old = V_new; }
```

10. `if (`$det > 0$`) printf(`"Yes, F is satisfiable and has ", det, " truth assignments."`);`
11. `else printf(`"No, F is unsatisfiable."`); }`

`void` inc`(int` i, `list_of_clauses` lc, `int` $level$`) {`
12. `for (int` $j = i-1$; $j > 0$; j`--)`
13. `if (`$dif_{V_{new}}(lc, C_j)$ `!= 0) {`
14. `if` odd($level$) $det = det - dif_{V_{new}}(lc, C_j)$;
15. `else` $det = det + dif_{V_{new}}(lc, C_j)$;
16. $inc(j, [lc, C_j], level+1)$ `} }`

Let us point out (informally) the correctness and finiteness of Algorithm **B**. First, the value of det is 1 and the set of clauses will be processed one by one according to the **for** statement between lines **2** and **9**. The lines **3** and **4** underline item a) of Lemma 3.1. If the new clause contains more variables, then det will be multiplied with $2^{|V(C_i)-V_{old}|}$ at line **6** (this is due to item b) of Lemma 3.1). Then, at lines **7** and **8**, we add the **increment** at the old value of the determinant (this is correct due to Theorem 4.1). Actually, line **8** corresponds to item c) of Corollary 4.2.

The procedure `inc`(i, lc, $level$) computes the **increment** value of the corresponding clausal sub-tree with root C_i, starting from the current $level$. The correctness follows from Notation 4.1 (lines **12** to **16**) and Remark 3.1 (line **13**). The second argument (i.e. lc) is needed for keeping the path between the root and the current clause. In general, this list is $[C_i, C_{j_1}, ..., C_{j_s}]$, where $j_k \in \{i-1, ..., 1, 0\}$ for any $k \in \{1, ..., s\}$ (the case $s = 0$ corresponds to the list $[C_i]$). We denoted the list data structure using square brackets, so, $[lc, C_j]$ means the list obtained by catenating the list lc with $[C_j]$.

The finiteness of the recursive procedure `inc()` is also obvious, knowing that it corresponds to a depth-first traversal of a finite tree. Actually, by joining all the clausal sub-trees corresponding to the execution of `inc()`, we get an isomorphic tree with $CT_{red}(F)$. This is due to Lemma 4.1 which allows the re-arrangement of the nodes of $CT_{red}(F)$. That is, the direct descendants of a node will not be labelled in the ascending order (i.e. $C_1, C_2, ..., C_l$) like in $CT_{red}(F)$, but in the descending order. This is actually the *key ingredient* of the incremental computation of the determinant of a clausal formula.

V. Incremental Approach for the Verification of a Real-Time System

According to our notations, the condition of **while** statement of Algorithm **A** can be rewritten as $det_V(SP_k \wedge \neg SA_k) > 0$. This condition can be efficiently evaluated in an incremental way using Algorithm **B**, based on the value of $det_V(SP_{k-1} \wedge \neg SA_{k-1})$, for any $k > 1$. The steps **3, 4, 5** of Algorithm **A** can be done by analysing the old and new constraints graphs, according to the new clauses of $SP_k \wedge \neg SA_k$.

In order to see how incremental computing is used, we suppose that new events can be added to a given real-time system. For our study-case, let us consider two new events: $CarCrossingLeft$ and $CarCrossingRight$ (denoted shortly as CCL and CCR), with the following additional behavioral specification: "A car crossing from the left **or** right needs at most 10 seconds to cross the railroad" and at the safety assertion: "**If** the train starts crossing the railroad crossing, there were no cars crossing neither from left **nor** from the right in the last 5 seconds". In path RTL, this is expresses as: $\forall z_1$ @($\downarrow CCL$, z_1)$-$ 10 $\leq$ @($\uparrow CCL$, z_1) $\wedge$ $\forall z_2$ @($\downarrow CCR$, z_2)$-$ 10 $\leq$ @($\uparrow CCR$, z_2). The other sentence is translated into: $\forall v_1$ @($\downarrow CCL$, v_1)$+$ 5 $\leq$ @($\uparrow TrainCrossing$, u) $\wedge$ $\forall v_2$ @($\downarrow CCR$, v_2)$+$ 5 $\leq$ @($\uparrow TrainCrossing$, u).

This time we denote the newly obtained $SP \wedge \neg SA$ with F_2. As we can see from Figure 3, the new constraint graph has the same positive cycles as the one from Figure 1. Summarizing, $F_2 = \{ \{A_1\}, \{A_2\}, \{A_3\}, \{A_4\}, \{A_5\}, \{A_6, A_7, A_{10}, A_{11}\}, \{A_8\}, \{A_9\}, \{\overline{A_2}, \overline{A_4}, \overline{A_6}\}, \{\overline{A_4}, \overline{A_5}, \overline{A_6}, \overline{A_7}\}, \{\overline{A_1}, \overline{A_3}, \overline{A_5}, \overline{A_7}\} \}$ over the set of variables $V_2 = \{A_1, ..., A_{11}\}$.

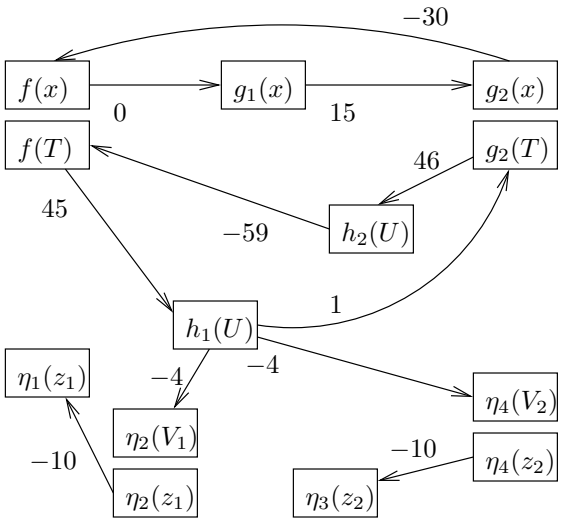

Figure 3. Railroad crossing constraint graph (2)

To illustrate the idea of incremental computing of the determinant, we would like to compute $det_{V_2}(F_2)$ using $det_{V_1}(F_1)$. It is obvious that, compared to F_1, the formula F_2 has four more variables, two new clauses, and one clause modified. We may choose to do first either incremental computing or decremental computing. For simplicity, we apply first Corollary 4.1 and then Theorem 4.1. It follows that $det_{V_2}(F_2) = det_{V_2}(F_1) + inc_{V_2}(\{A_8\}, F_1) + inc_{V_2}(\{A_9\}, F_1 \cup \{\{A_8\}\}) + inc_{V_2}(\{A_6, A_7, A_{10}, A_{11}\}, F_1 \cup \{\{A_8\}\} \cup \{\{A_9\}\}) - inc_{V_2}(\{A_6, A_7\}, F_1 \cup \{\{A_8\}\} \cup \{\{A_9\}\} \cup \{\{A_6, A_7, A_{10}, A_{11}\}\})$

Instead of calling procedure inc`()`, Corollary 4.2 will be applied:

b1) From item c) of Lemma 3.1, it follows that $det_{V_2}(F_1) = 0$. Now, applying one of the items a) or d) of Corollary 4.2, it follows that $inc_{V_2}(\{A_8\}, F_1) = 0$ and $inc_{V_2}(\{A_9\}, F_1 \cup \{\{A_8\}\}) = 0$;

b2) Because $\{A_6, A_7\} \subseteq \{A_6, A_7, A_{10}, A_{11}\}$, according to item c) of Corollary 4.2, it follows that $inc_{V_2}(\{A_6, A_7, A_{10}, A_{11}\}, F_1 \cup \{\{A_8\}\} \cup \{\{A_9\}\}) = 0$;

b3) We call procedure inc`()` only for $inc_{V_2}(\{A_6, A_7\}, F_1 \cup \{\{A_8\}\} \cup \{\{A_9\}\} \cup \{\{A_6, A_7, A_{10}, A_{11}\}\})$ getting -3 (generating 267 nodes in the associated clausal tree).

So $det_{V_2}(F_2) = 3$, which means that F_2 is satisfiable, hence the real-time system is unsafe. In our attempt at a systematic debugging of the real-time system, it is easy to see that:

1. It is good to have at least one more negative clause, so these correspond to at least one more positive cycle;

2. This cycle have to contain some of the new literals, namely from $V_2 - V_1$.

Looking at Figure 3, the constraint graph has six new nodes (two pairs of nodes can be considered single nodes due to the unification process of the first-order logic). In order to minimize the determinant, some (negative) clauses should be discovered, i.e. these new nodes should be involved in a positive cycle. For instance, the starting node of the positive cycle can be considered as the one labelled with $h_1(U)$. It must be possible to continue the path from the nodes labelled with $\eta_1(z_1)$ and $\eta_3(z_2)$, respectively. Going back to the safety assertion, we may add "**If** the gate starts to go down, then no car from the left **and** the right will start to cross the railroad". In path RTL, this is equivalent to saying: $@(\uparrow CCL,\ v_1) \leq @(\uparrow DownGate,\ t)$ and $@(\uparrow CCR,\ v_2) \leq @(\uparrow DownGate,\ t)$.

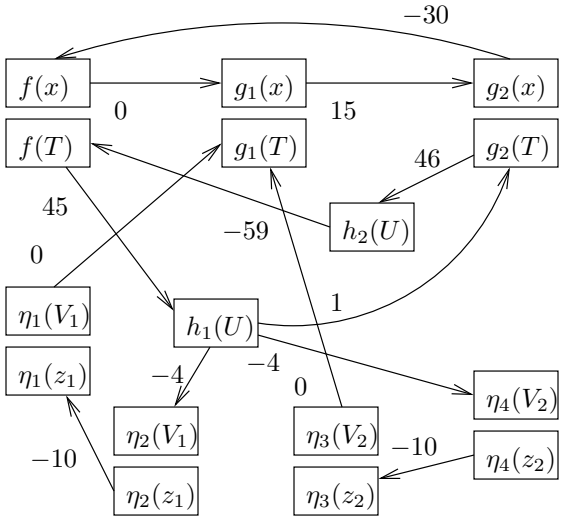

Figure 4. Railroad crossing constraint graph (3)

With the two new negative clauses, two new positive cycles (Figure 4) will be generated, namely: $C_{14} = \{\overline{A_2},\ \overline{A_3},\ \overline{A_4},\ \overline{A_9},\ \overline{A_{11}},\ \overline{A_{13}}\}$ and $C_{15} = \{\overline{A_2},\ \overline{A_3},\ \overline{A_4},\ \overline{A_8},\ \overline{A_{10}},\ \overline{A_{12}}\}$. Let us denote $C_{12} = \{A_{12}\}$, $C_{13} = \{A_{13}\}$, and $V_3 = \{A_1,\ ...,\ A_{13}\}$, and the new clausal formula $F_3 = F_2 \cup \{C_{12}, C_{13}, C_{14}, C_{15}\}$. This time, only incremental computing theorem is used since we have only added some clauses. Applying Theorem 4.1, we get $det_{V_3}(F_3) = det_{V_3}(F_2) + inc_{V_3}(C_{12},\ F_2) + inc_{V_3}(C_{13},\ F_2 \cup \{C_{12}\}) + inc_{V_3}(C_{14},\ F_2 \cup \{C_{12}\} \cup \{C_{13}\}) + inc_{V_3}(C_{15},\ F_2 \cup \{C_{12}\} \cup \{C_{13}\} \cup \{C_{14}\})$. According to item c) of Lemma 3.1, because $|V_3| = |V_2| + 2$ (two new variables were added), it follows that $det_{V_3}(F_3) = 2^2 \cdot det_{V_3}(F_2) = 12$. Computing the four increments involved, we get $det_{V_3}(F_3) = 12 - 6 - 3 - 2 - 1 = 0$. This means that F_3 is unsatisfiable, and hence the safety of the revisited solution.

VI. Experimental results of Algorithm B

This subsection is devoted to an efficient implementation, as well as the experimental results of our incremental approach.

Item b) of Theorem 4.1 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. However, in the worst case, addition a new clause can double the number of new nodes corresponding to the incremental tree. For instance, consider $F = \{\{A_1\}, \{A_2\},\ ...,\ \{A_n\}\}$ over $V = \{A_1,\ ...,\ A_n\}$, then $CIT_{red}(\{A_{n+1}\},\ F)$ has 2^n nodes. On the other hand, if F is $\{C_1,\ ...,\ C_l\}$, the best case for incremental algorithm to compute $inc(C, F)$ will be when $\forall\ i \in \{1,\ ...,\ l\},\ \exists\ L_i \in C$, such that $\overline{L_i} \in C_i$. In this case, the tree $CIT_{red}(C, F)$ has only one node, that is the one labelled with $(C,\ dif_V(C))$.

The practical efficiency of the algorithm can be improved by adopting the numerical coding. First, we will not actually "create" any node of the 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. For instance, the boolean formula associated to the X-38 system has about 50 variables and 100 clauses [6]. So, the variable det can be implemented as an integer or boolean array, knowing that its value is between 0 and $2^{|V|}$ (details in [14]).

The improvements of Algorithm **B** can be considered:

a) Algorithm **B** refers to the addition of only one clause at a time. According to Theorem 4.1, it is possible to deal with the new clauses in parallel (by treating all of new clauses at the same time, and not sequentially).

b) Algorithm **B** works only for adding clauses, but not for their removing. However, Algorithm **B** can be easily adapted to deal with the removal of the clauses using Corollary 4.1.

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 on the time spend 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_{red}^{new}, $CIT_{red}(C_{l+1}, F)$ by CIT_{red}^1, and $CIT_{red}(C_{l+2}, F \cup \{C_{l+1}\})$ by CIT_{red}^2. Our testing instances refer to different values for (n, l).

(n, l)	CT_{red}^{new}		$CT_{red}(F)$	
	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 1. The non-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

Table 2. The incremental approach

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 $0.16 \approx 0.06 + 0.01 + 0.05$. 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).

VII. Related Work

As stated in [15], the SAT problem has a special interest in the artificial intelligence community because of its relationship to deductive reasoning. The #SAT problem is a valuable approach for evaluating techniques in an effort to avoid computational difficulties, such as the constraint satisfaction and the knowledge compilation.

Model checking ([16], [17]) is an important technique for verifying sequential design. In model checking, the specification of a design is expressed in temporal logic and the implementation is described as a finite state machine. Model checking using ordered binary decision diagrams [18], denoted by OBDDs, is called symbolic model checking ([19], [20]). With the introduction of bounded model checking [21], efficient propositional decision procedures for symbolic model checking begin to appear. In bounded model checking, only paths of bounded length k are considered. A comparison between BDD and SAT solvers has been done in [22] and to our counting strategy in [14]. Briefly, if a given problem requests not only the number of truth assignments, but also the assignments themselves, then OBDDs may be more useful than the determinant of F. On the other hand, for a given clausal formula F, there is only one $CT_{red}(F)$ and even if the clauses are re-ordered, the size of the associated clausal tree is the same [14]. In constrast, for a given F, there may be many (i.e. an exponential number depending on the number of variables) associated OBDDs, and the problem of finding the best reordering for representing F as an OBDD is coNP [18].

VIII. Conclusions and Future Work

In this paper, we embedded the incremental computation of the determinant of a clausal formula in the verification of timing constraints of a real-time system. We considered the well-known example of the railroad crossing. The debugging of the new specification can be done manually. An open problem is to do this process in an automatic way when analysing the constraint graph.

We thank to the unknown referees for their very useful remarks, suggestions and comments which improved the paper.

References

[1] F. Jahanian and A. Mok, "A graph-theoretic approach for timing analysis and its implementation," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 961–975, 1987.

[2] F. Jahanian and D. A. Stuart, "A method for verifying properties of modechart specifications," in *Proceedings of 9-th IEEE Real-Time Systems Symposium*, 1988, pp. 12–21.

[3] A. M. K. Cheng, *Real-time systems. Scheduling, Analysis, and Verification*. U. S. A.: Wiley-Interscience, 2002.

[4] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 890–904, 1986.

[5] F. Wang and A. K. Mok, "RTL and refutation by positive cycles," in *Proceedings of Formal Methods Europe Symposium*, ser. Lecture Notes in Computer Science, vol. 873, Springer Verlag, 1994, pp. 659–680.

[6] L. E. P. Rice and A. M. K. Cheng, "Timing analysis of the X-38 space station crew return vehicle avionics," in *Proceedings of the 5-th IEEE-CS Real-Time Technology and Applications Symposium*, 1999, pp. 255–264.

[7] A. M. K. Cheng, Apr. 2003, private communication.

[8] J. N. Hooker, "Solving the incremental satisfiability problem," *J. Logic Programming*, vol. 15, pp. 177–186, 1993.

[9] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.

[10] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proceedings of the 38th ACM/IEEE Conference on Design Automation*, 2001, pp. 542–545.

[11] S. Andrei, "The determinant of the boolean formulae," *Analele Universităţii Bucureşti, Informatică*, vol. XLIV, pp. 83–92, 1995.

[12] K. Iwana, "CNF satisfiability test by counting and polynomial average time," *Siam J. Comput.*, vol. 18, no. 2, pp. 385–391, 1989.

[13] E. Birbaum and E. L. Lozinskii, "The good old Davis-Putnam procedure helps counting models," *Journal of Artificial Intelligence Research*, vol. 10, pp. 457–477, 1999.

[14] S. Andrei and W.-N. Chin, "Incremental satisfiability counting for real-time systems," Faculty of Computer Science, Iaşi University, România, Tech. Rep. TR03-04, Sept. 2003. [Online]. Available: http://www.infoiasi.ro/tr/tr.pl.cgi

[15] D. Roth, "On the hardness of approximate reasoning," *Artificial Intelligence*, vol. 82, pp. 273–302, 1996.

[16] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the IBM Workshop on Logics of Programs*, ser. Lecture Notes in Computer Science, vol. 131, Springer Verlag, 1981, pp. 52–71.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[18] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[19] J. R. Burch, E. M. Clarke, and K. L. McMillan, "Symbolic model checking: 10^{20} states ans beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

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

[21] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of TACAS'99*, ser. Lecture Notes in Computer Science, vol. 1579, Springer Verlag, 1999, pp. 193–207.

[22] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, 1999, pp. 317–320.

Ştefan Andrei is a Research Fellow in the National University of Singapore under the Singapore-MIT Alliance (SMA). He has received his B.Sc. and M.Sc. in Computer Science, in 1994 and 1995, respectively, from Iaşi University, România and PhD in Computer Science in 2000 from the Hamburg University, Germany. Between 1997 and 2000, he got the following academic awards: DAAD scholarship, TEMPUS S_JEP 11168-96 scholarship and World Bank Joint Japan Graduate Scholarship at Fachbereich Informatik, Hamburg Universitaet, Germany. He is currently working on formal languages, compilers and real-time systems. More details about him can be found at `http://www.infoiasi.ro/~stefan`

Wei Ngan Chin is an Associate Professor at the Department of Computer Science, School of Computing, National University of Singapore, and a Fellow in the Computer Science Programme of the Singapore-MIT Alliance. He has received his B.Sc. and M.Sc. in Computer Science, in 1982 and 1983, respectively, from University of Manchester, United Kingdom, and PhD in Computing, in 1990 from the Imperial College of Science, Technology and Medicine, United Kingdom. His current research interests are functional programming, program transformation, parallel systems, software models and methods. More details about him can be found at `http://www.comp.nus.edu.sg/~chinwn/`

Martin Rinard received the Sc.B. in Computer Science, Magna cum Laude and with Honors, from Brown University in 1984. He received the Ph.D. in Computer Science from Stanford University in 1994. He joined the Computer Science Department at the University of California, Santa Barbara as an Assistant Professor in 1994, then moved to MIT as an Assistant Professor in 1997. Since 2000, he is an Associate Professor at MIT. His current research interests are pointer and escape analysis, automatic parallelization, commutativity analysis, symbolic analysis of divide and conquer algorithms, synchronization optimizations, and credible compilation. More details about him can be found at `http://cag.csail.mit.edu/~rinard/`