Verification of Timed Automata via Satisfiability Checking*

Peter Niebert[2], Moez Mahfoudh[1], Eugene Asarin[1], Marius Bozga[1], Oded Maler[1] and Navendu Jain[3]

[1] VERIMAG, 2 Av. de Vignate, 38610 Gières, France
[Moez.Mahfoudh | Eugene.Asarin | Marius.Bozga | Oded.Maler]@imag.fr
[2] Laboratoire d'Informatique Fondamentale, CMI, 39 rue Joliot-Curie
13453 Marseille Cedex 13, France, niebert@cmi.univ-mrs.fr
[3] Computer Science and Eng. Dept., Indian Inst. of Technology
Hauz Khas, New Delhi, India, navendu@cse.iitd.ac.in

Abstract. In this paper we show how to translate bounded-length verification problems for timed automata into formulae in *difference logic*, a propositional logic enriched with timing constraints. We describe the principles of a satisfiability checker specialized for this logic that we have implemented and report some preliminary experimental results.

1 Introduction

The generic problem of verification can be phrased as follows: given a description of a transition system, check whether its set of possible behaviors contains a behavior violating some desired property. It is known for quite a while that the problem is decidable for finite-state systems and various specification formalisms [QS81,EC82,LP84,VW86] via graph search algorithms. However, most systems of interest are given as a composition of many interacting sub-systems and their global state-space is prohibitively-large for enumerative graph algorithms. To treat such systems one needs symbolic techniques that perform the reachability computation on an implicit syntactic representation of the system. This is the basis of what is called *symbolic model-checking* [McM93,BCM+93].

To be more concrete, consider a system over a set $\mathcal{B}$ of state variables and its transition relation $R(\mathcal{B}, \mathcal{B}')$, written using an auxiliary set of "next-state" variables $\mathcal{B}'$. The global transition relation is expressed as a conjunction of the local transition formulae for the system components:

$$R(\mathcal{B}, \mathcal{B}') = R_1(\mathcal{B}, \mathcal{B}') \wedge R_2(\mathcal{B}, \mathcal{B}') \wedge \ldots \wedge R_n(\mathcal{B}, \mathcal{B}').$$

The standard algorithm for checking whether from a set of initial states F one can reach a set G works as follows:

* This work was partially supported by the EC project IST-2001-35302 AMETIST (Advanced Methods for Timed Systems).

repeat
  $F(\mathcal{B}) := \exists \mathcal{B}' [\, F(\mathcal{B}') \wedge R(\mathcal{B}', \mathcal{B})]$
until $F(\mathcal{B}) \wedge G(\mathcal{B}) \neq \emptyset$

At every iteration k of the algorithm, the formula F characterizes the states reachable from the initial set within exactly k steps. The Boolean operations and the $\exists$-operation are implemented usually using BDDs, a canonical representation for propositional formulae [Bry86]. A variant of the algorithm which keeps a representation of all states reachable within at most k steps is guaranteed to terminate after finitely many steps.

In recent years it has been realized that this way of solving the reachability problem is not always the most efficient, and that the representation of intermediate sets of states by BDDs can explode in size. Instead of an alternating sequence of next-state computation and elimination of intermediate states, one can construct a formula expressing the existence of a path of length k, and then apply generic SAT solvers to the formula [SS90,BCCZ99,BCCZ99]. The formula looks like this:

$$\exists \mathcal{B}^0, \ldots, \exists \mathcal{B}^k \; F(\mathcal{B}^0) \wedge R(\mathcal{B}^0, \mathcal{B}^1) \wedge R(\mathcal{B}^1, \mathcal{B}^2) \wedge \ldots \wedge R(\mathcal{B}^{k-1}, \mathcal{B}^k) \wedge G(\mathcal{B}^k).$$

The advantage in using a SAT solver for such a formula lies in the fact that we are not restricted anymore to a fixed order of variable elimination and can use any of the numerous techniques for solving the satisfiability problem (which is, perhaps, the most celebrated discrete computational problem). Of course, this idea is most useful for systems that do exhibit undesired behavior, and is less so for correct systems of large diameter, but bug hunting is a respectable activity especially for systems too large for complete verification.

In this work we take this idea further and apply it to *timed systems* where the computational difficulty is much bigger. To this end we define *difference logic*, a propositional logic augmented with numerical variables and difference constraints between pairs of such variables. We then define a procedure for expressing bounded reachability problems for timed automata as formulae in this logic. These formulae are then checked for satisfiability by a new SAT solver for difference logic that we have designed and implemented. Preliminary experimental results are reported.

The rest of the paper is structured as follows. In section 2 we define difference logic and the conjunctive normal form for its formulae. The derivation of formulae for bounded reachability problems for "flat" timed automata is described in Section 3 followed by a compositional version of this transalation in Section 4. Section 5 is dedicated to the description of the solver, while some preliminary experimental results are described in Section 6. We conclude with some related work.

2 Difference Logic

In this section we define the class of formulae that we consider. This class has already been used elsewhere (see e.g. [MLAH99,LPWY99,ABK+97,BM00,ACG99]) and is probably part of folklore, but to give it a name let us call it *difference logic* (DL). We use two slightly different variants, DLZ and DLR depending on the domain of the numerical variables.

Definition 1 (Difference Logic). *Let $\mathcal{B} = \{B_1, B_2, \dots\}$ be a set of Boolean variables and $\mathcal{X} = \{X_1, X_2, \dots\}$ be a set of numerical variables. The set of atomic formulae of $DL(\mathcal{B}, \mathcal{X})$ consists of the Boolean variables in $\mathcal{B}$ and numerical constraints of the following forms:*

$$X_i - X_j \geq c$$

(with $c \in \mathbb{Z}$ for DLZ and $c \in \mathbb{Q}$ for DLR) and

$$X_i - X_j > c$$

with $c \in \mathbb{Q}$ for DLR only.[1]

The set $\mathcal{F}$ of all DL formulae is the smallest set containing the atomic formulae which is closed under negation ($\varphi \in \mathcal{F}$ implies $\neg\varphi \in \mathcal{F}$) and a collection of binary Boolean connectives ($\varphi_1, \varphi_2 \in \mathcal{F}$ implies $\varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2, \varphi_1 \rightarrow \varphi_2, \dots \in \mathcal{F}$).

An $(\mathcal{X}, \mathcal{B})$-*valuation* consists of two functions (overloaded with the name v) $v : \mathcal{B} \rightarrow \{T, F\}$ and $v : \mathcal{X} \rightarrow \mathbb{Z}$ for DLZ or $v : \mathcal{X} \rightarrow \mathbb{R}$ for DLR. The valuation v is extended to all $DL(\mathcal{B}, \mathcal{X})$ formulae by letting

$$v(X_i - X_j \geq c) = T \text{ iff } v(X_i) - v(X_j) \geq c$$

and applying the obvious rules for the Boolean connectives.

A formula φ is satisfied by a valuation v iff $v(\varphi) = T$ (we denote it also by $v \models \varphi$). A formula φ is satisfiable if it has a satisfying valuation.

Proposition 1. *The satisfiability problem for $DL(\mathcal{B}, \mathcal{X})$ is NP-complete.*

Proof. NP-hardness is an immediate consequence of the Boolean case, Cook's theorem [GJ79]. For NP-easiness, a non-deterministic algorithm works by guessing which atomic formulae (Boolean variables and inequalities) appearing in the formula are true and which are false. A polynomial time test then has to check that this assignment renders the entire formula true (linear time in the size of the formula) and that the corresponding set of constraints on the reals is in fact satisfiable. The satisfiability of a *conjunction* of difference constraints (a special case of linear programming) can be solved in polynomial (cubic) time using a variant of the Floyd-Warshall algorithm [CLRS01]. □

We work with formulae in conjunctive normal with at most 3 literals in a clause.

Definition 2 (3CNF). *A Boolean literal is a formula of the form B or $\neg B$ with $B \in \mathcal{B}$. A numerical literal is a formula of the form $X - Y \geq c$ (also $X - Y > c$ for DLR). A 3-clause is a disjunction $C = L_1 \vee L_2 \vee L_3$ of at most 3 literals* at most one of which is a numerical literal. *A formula of difference logic is in 3CNF if it is a conjunction $\bigwedge_{k \in K} C_k$ of a set of 3-clauses C_k.*

[1] Over integers, the constraint $X - Y > c$ is of course equivalent to $X - Y \geq c + 1$, hence $>$-constraints are obsolete in the integer case and life is easier without them.

The restriction to at most one numerical literal per clause plays an important role in the implementation of our procedure. The restriction to 3-clauses is less pertinent — it simplifies the description of proof methods based on the Davis-Putnam approach. We have developed another version with unbounded clause size.

As in propositional logic (see [T70]), translations from arbitrary formulae to 3CNF need not be costly if auxiliary Boolean variables are introduced:

Proposition 2. *From an arbitrary $DL(\mathcal{B}, \mathcal{X})$ formula φ one can derive a $DL(\mathcal{B}', \mathcal{X})$ formula φ' over an extended set of Boolean variables $\mathcal{B}' \supseteq \mathcal{B}$, such that*

- *φ' is in 3CNF;*
- *any $(\mathcal{B}, \mathcal{X})$-valuation v satisfying φ can be extended to a $(\mathcal{B}', \mathcal{X})$-valuation v' satisfying φ'.*
- *(the projection of) any valuation v' satisfying φ' satisfies equally φ.*
- *$|\varphi'| = O(|\varphi|)$.*

Proof. (Sketch) A simple construction to achieve this is to introduce for each composed sub-formula ψ of φ a fresh variable B_ψ to express whether for a given valuation ψ holds. Then the structure of φ can be broken up into local semantic constraints by coding, using 3CNF clauses, the semantic relation of ψ with its immediate sub-formulae. For instance, if $\psi = \psi_1 \vee \psi_2$ then we will add the three clauses $\{\neg B_\psi \vee B_{\psi_1} \vee B_{\psi_2}, B_\psi \vee \neg B_{\psi_1}, B_\psi \vee \neg B_{\psi_2}\}$ to our set of clauses. In addition to these structural clauses, a clause B_φ is used to express that φ should be true. It is easy to extend a satisfying valuation of φ to a satisfying valuation for the set of clauses constructed this way and, conversely, projecting the additional variables of a satisfying valuation of φ' yields a satisfying valuation of φ. □

As a tiny example, consider the formula φ consisting of a clause

$$(Y - Z \geq -2 \vee Z - X \geq 1)$$

with two numerical literals. To transform it to 3CNF we use an additional Boolean variable B to represent $Y - Z \geq -2$. Using the fact that $B \leftrightarrow Y - Z \geq -2$ can be written as

$$(B \leftarrow Y - Z \geq -2) \wedge (B \rightarrow Y - Z \geq -2)$$

we get

$$(B \vee Z - X \geq 1) \wedge (\neg B \vee Y - Z \geq -2) \wedge (B \vee Z - Y > 2).$$

In practice, optimized translations introducing less variables and with certain additional structural properties are possible (see [GW99] for a more detailed discussion).

3 From Flat Timed Automata to Difference Logic

Timed automata [AD94] are automata augmented with clock variables. Their behavior consists of an alternation of discrete transitions and passage of time where the automtaton stays in a state and the clock values grow with a uniform rate. Clock values can enable transitions, disable staying in a state and be reset by transitions. We will start with "flat" timed automata and define their translation to DL, and later will proceed to a product of interacting automata. A similar translation, proposed in [HNSY94], is the basis for the algorithmic verification of timed automata. An integer bounded inequality over a set $\mathcal{C}$ of variables is either $C \leq d$, $C < d$, $C \geq d$ or $C > d$ for $C \in \mathcal{C}$ and an integer constant d.

Definition 3 (Timed Automaton). *A timed automaton is a tuple $\mathcal{A} = (Q, \mathcal{C}, S, \Delta)$ where Q is a finite set of states, $\mathcal{C}$ is a finite set of clock variables, S is a function which associates with every state q a conjunction S_q of integer bounded inequalities over $\mathcal{C}$ (staying conditions), and Δ is a transition relation consisting of tuples of the form (q, g, ρ, q') where q and q' are states, $\rho \subseteq \mathcal{C}$ and g (the transition guard) is a conjunction of integer bounded inequalities over $\mathcal{C}$.*

A *clock valuation* is a function $\mathbf{v} : \mathcal{C} \to \mathbb{R}_+$, or equivalently a $|\mathcal{C}|$-dimensional vector over $\mathbb{R}_+$. We denote the set of all clock valuations by $\mathcal{H}$. A configuration of the automaton is hence a pair $(q, \mathbf{v}) \in Q \times \mathcal{H}$ consisting of a discrete state (sometimes called "location") and a clock valuation. Every subset $\rho \subseteq \mathcal{C}$ induces a reset function $\mathrm{Reset}_\rho : \mathcal{H} \to \mathcal{H}$ defined for every clock valuation $\mathbf{v}$ and every clock variable $C \in \mathcal{C}$ as

$$\mathrm{Reset}_\rho\, \mathbf{v}(C) = \begin{cases} 0 & \text{if } C \in \rho \\ \mathbf{v}(C) & \text{if } C \notin \rho \end{cases}$$

That is, Reset_ρ resets to zero all the clocks in ρ and leaves the other clocks unchanged. We use $\mathbf{1}$ to denote the unit vector $(1, \ldots, 1)$ and $\mathbf{0}$ for the zero vector.

Definition 4 (Steps and Runs). *A step of the automaton is one of the following:*

- *A discrete step:*

$$(q, \mathbf{v}) \xrightarrow{\delta} (q', \mathbf{v}'),$$

  where $\delta = (q, g, \rho, q') \in \Delta$, such that $\mathbf{v}$ satisfies g and $\mathbf{v}' = \mathrm{Reset}_\rho(\mathbf{v})$.
- *A time step:*

$$(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1}),$$

  where $t \geq 0$, and $\mathbf{v} + t\mathbf{1}$ satisfies S_q.

A finite run of a timed automaton is a finite sequence of steps

$$(q_0, \mathbf{v}_0) \xrightarrow{z_1} (q_1, \mathbf{v}_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, \mathbf{v}_n).$$

Steps and runs can be extended to include *time stamps* that indicate the absolute time since the beginning of the run. This can be viewed as having an additional clock which is never reset to zero. An extended discrete step is thus

$$(q, \mathbf{v}, T) \xrightarrow{\delta} (q', \mathbf{v}', T),$$

and an extended time step is

$$(q, \mathbf{v}, T) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1}, T + t).$$

Note that a single behavior of the automaton can be represented by infinitely many runs due to splitting of time steps. For example, the step above can be split into

$$(q, \mathbf{v}, T) \xrightarrow{t'} (q, \mathbf{v} + t'\mathbf{1}, T + t') \xrightarrow{t-t'} (q, \mathbf{v} + t\mathbf{1}, T + t).$$

In particular the definition of a time step allows $t = 0$ which means that idle transitions that do nothing and take no time can be inserted anywhere inside a run. This will be used in the sequel. We will refer to a run in which there are no two consecutive time steps as a *minimal run*.

To build a DL formula that characterizes valid runs of the automaton we assume first that Q can be encoded by a set $\mathcal{B}$ of Boolean variables so that $\Phi_q(\mathcal{B})$ is the formula over those variables denoting a state q. Such state formulae can be extended using disjunction into formulae of the form Φ_P for every $P \subseteq Q$. In order to express step formula we will use auxiliary sets of variables $\mathcal{B}'$ and $\mathcal{C}'$ to denote the values of state and clock variables after the step.

The formula $\Phi_\rho(\mathcal{C}, \mathcal{C}')$ expressing the effect of applying Reset_ρ is

$$\Phi_\rho(\mathcal{C}, \mathcal{C}') \equiv \bigwedge_{C_m \in \rho} C'_m = 0 \wedge \bigwedge_{C_m \notin \rho} C'_m = C_m.$$

The formula $\Phi_\tau(\mathcal{C}, \mathcal{C}')$ for the passage of time is

$$\Phi_\tau(\mathcal{C}, \mathcal{C}') \equiv \exists t \; t \geq 0 \wedge \bigwedge_m C'_m - C_m = t.$$

The formula $\Psi_\delta(\mathcal{B}, \mathcal{C}, \mathcal{B}', \mathcal{C}')$ for a step associated with a transition $\delta = (q, g, \rho, q')$ is

$$\Psi_\delta(\mathcal{B}, \mathcal{C}, \mathcal{B}', \mathcal{C}') \equiv \Phi_q(\mathcal{B}) \wedge \Phi_g(\mathcal{C}) \wedge \Phi_\rho(\mathcal{C}, \mathcal{C}') \wedge \Phi_{q'}(\mathcal{B}')$$

where $\Phi_g(\mathcal{C})$ is just the substitution of $\mathcal{C}$ in the guard g. The formula $\Psi_q(\mathcal{B}, \mathcal{C}, \mathcal{B}', \mathcal{C}')$ for a time step at state q is

$$\Psi_q(\mathcal{B}, \mathcal{C}, \mathcal{B}', \mathcal{C}') \equiv \Phi_q(\mathcal{B}) \wedge \Phi_\tau(\mathcal{C}, \mathcal{C}') \wedge \Phi_{s,q}(\mathcal{C}') \wedge \Phi_q(\mathcal{B}')$$

where $\Phi_{s,q}(\mathcal{C}')$ is just the substitution of $\mathcal{C}'$ in S_q. The formula for a valid step is

$$\Psi(\mathcal{B}, \mathcal{C}, \mathcal{B}', \mathcal{C}') \equiv \bigvee_{q \in Q} \Psi_q \vee \bigvee_{\delta \in \Delta} \Psi_\delta.$$

The formula $\Psi_k(\mathcal{B}^0, \mathcal{C}^0, \ldots, \mathcal{B}^k, \mathcal{C}^k)$ characterizing a run of length k is written as

$$\Psi_k \equiv \Psi(\mathcal{B}^0, \mathcal{C}^0, \mathcal{B}^1, \mathcal{C}^1) \wedge \Psi(\mathcal{B}^1, \mathcal{C}^1, \mathcal{B}^2, \mathcal{C}^2) \wedge \ldots \wedge \Psi(\mathcal{B}^{k-1}, \mathcal{C}^{k-1}, \mathcal{B}^k, \mathcal{C}^k).$$

Due to idling this formula is satisfied also by runs whose minimal run is of length smaller than k. Finally a run of length k starting from a set G and ending in a set H is written as

$$\Phi_G(\mathcal{B}^0, \mathcal{C}^0) \wedge \Psi_k(\mathcal{B}^0, \mathcal{C}^0, \ldots, \mathcal{B}^k, \mathcal{C}^k) \wedge \Phi_H(\mathcal{B}^k, \mathcal{C}^k).$$

As the alert reader might have noticed, the inequalities in Φ_τ are outside the scope of DL. After eliminating t we obtain

$$\Phi_\tau(\mathcal{C}, \mathcal{C}') \equiv \bigwedge_i \bigwedge_{j \neq i} (C'_i - C_i) = (C'_j - C_j) \geq 0$$

with numerical constraints that relate 4 numerical variables. This can be, however, circumvented by a change of variables which has an intuitive meaning (see also [BJLY98]). Consider a state extended with a time-stamp T. For every clock C_i, the variable $X_i = T - \mathcal{C}_i$ represents the last time when C_i was reset (we use the notation $\mathcal{X} = T - \mathcal{C}$ for the whole transformation). It is not hard to see that an $(\mathcal{X}, T)$-valuation gives a state representation "isomorphic" to a $\mathcal{C}$-valuation: the guard and staying conditions should be evaluated on $T - \mathcal{X}$ instead of on $\mathcal{C}$, passage of time affects only T while a reset of C_i at time T corresponds to the assignment $X_i := T$. All the above formulae can be transformed into formulae in $\mathcal{X}$ and T as follows. The reset formula becomes

$$\Phi_\rho(\mathcal{X}, \mathcal{X}', T) \equiv \bigwedge_{C_m \in \rho} X'_m = T \wedge \bigwedge_{C_m \notin \rho} X'_m = X_m.$$

Time passage is written as

$$\Phi_\tau(\mathcal{X}, T, \mathcal{X}', T') \equiv \bigwedge_m X'_m = X_m \wedge \exists t\, t \geq 0 \wedge T' - T = t$$

and after elimination of t as

$$\Phi_\tau(\mathcal{X}, T, \mathcal{X}', T') \equiv \bigwedge_m X'_m = X_m \wedge T' - T \geq 0.$$

The transition formula becomes

$$\Psi_\delta(\mathcal{B}, \mathcal{X}, T, \mathcal{B}', \mathcal{X}', T') \equiv \Phi_q(\mathcal{B}) \wedge \Phi_g(\mathcal{X}, T) \wedge \Phi_\rho(\mathcal{X}, \mathcal{X}', T) \wedge \Phi_{q'}(\mathcal{B}') \wedge T = T'$$

where $\Phi_g(\mathcal{X}, T)$ is the substitution of $T - \mathcal{X}$ in the guard g instead of $\mathcal{C}$. A time step at state q is expressed as

$$\Psi_q(\mathcal{B}, \mathcal{X}, T, \mathcal{B}', \mathcal{X}', T') \equiv \Phi_q(\mathcal{B}) \wedge \Phi_\tau(\mathcal{X}, T, \mathcal{X}', T') \wedge \Phi_{s,q}(\mathcal{X}', T') \wedge \Phi_q(\mathcal{B}')$$

where $\Phi_{s,q}(\mathcal{X}', T')$ is the substitution of $T' - \mathcal{X}'$ in S_q. The formula for a step is

$$\Psi(\mathcal{B}, \mathcal{X}, T, \mathcal{B}', \mathcal{X}', T') \equiv \bigvee_{q \in Q} \Psi_q \vee \bigvee_{\delta \in \Delta} \Psi_\delta.$$

The formula $\Psi_k(\mathcal{B}^0, \mathcal{X}^0, T^0, \ldots, \mathcal{B}^k, \mathcal{X}^k, T^k)$ for a run of length k is

$$\Psi_k \equiv \Psi(\mathcal{B}^0, \mathcal{X}^0, T^0, \mathcal{B}^1, \mathcal{X}^1, T^1) \wedge \ldots \wedge \Psi(\mathcal{B}^{k-1}, \mathcal{X}^{k-1}, T^{k-1}, \mathcal{B}^k, \mathcal{X}^k, T^k).$$

All these formulae are in DL and the price is the addition of k numerical variables that represent the dates at which the corresponding transitions were taken.

Proposition 3 (Translation). *Let $\mathcal{A}$ be a timed automaton with n states, m clocks and l transitions. The problem of reachability within at most k transitions can be expressed by a DL formula with $(k+1)\log n$ Boolean variables and $(k+1)(m+1)$ numerical variables. The size of the formula is $O(k(n+l)m^2 \log n)$.*

4 Compositional Translation

In this section we describe the construction of DL formulae for reachability problems for a product of interacting timed automata. There are many variants of composition operators and we will concentrate in the presentation on communication by variables, that is, any automaton may observe the state of the other automata and refer to their values in its transition guards and staying conditions. Consider several timed automata $\{\mathcal{A}_i\}_{i\in I}$ of the form $\mathcal{A}_i = (Q_i, \mathcal{C}_i, S_i, \Delta_i)$ running in parallel. We assume the states of each automaton $\mathcal{A}_i$ are encoded using a distinct set $\mathcal{B}_i$ of Boolean variables and let $\mathcal{B} = \bigcup \mathcal{B}_i$ and $\mathcal{C} = \bigcup \mathcal{C}_i$.

To express the mutual interaction between automata we use some notations. The set of automata whose state should be observed while taking the transition $\delta_i \in \Delta_i$ with a guard g_i is

$$J_{\delta_i} = \{j : Q_j \text{ appears in } g_i\}.$$

Likewise, the set of automata to be observed during time passage in a state $q_i \in Q_i$ is

$$J_{q_i} = \{j : Q_j \text{ appears in } S_{q_i}\}.$$

Finally the set of automata whose passage of time might be influenced by transitions in automaton $\mathcal{A}_i$ is

$$J'_i = \{j : \exists q_j \in Q_j \text{ s.t. } Q_i \text{ appears in } S_{q_j}\}.$$

A local discrete step of an automaton $\mathcal{A}_i$ is

$$(q_i, \mathbf{v}_i, T) \xrightarrow{\delta_i} (q'_i, \mathbf{v}'_i, T),$$

such that the transition guard g_i of $\delta_i \in \Delta_i$ is satisfied by the clocks of $\mathcal{A}_i$ *and* by the states of all other automata in J_{δ_i} at time T. A time step of $\mathcal{A}_i$ is

$$(q_i, \mathbf{v}_i, T) \xrightarrow{t} (q_i, \mathbf{v}_i + t\mathbf{1}, T + t).$$

such that S_{q_i} is satisfied by the clocks of $\mathcal{A}_i$ and by states of the automata in J_{q_i} during the interval $[T, T + t)$.

The first approach for constructing global runs and their corresponding formulae is rather straightforward. A global discrete step is of the form

$$\begin{array}{c}((q_1, \ldots, q_i, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}_i, \ldots, \mathbf{v}_n), T) \xrightarrow{\delta_i} \\ ((q_1, \ldots, q'_i, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}'_i, \ldots, \mathbf{v}_n), T)\end{array}$$

such that $(q_i, \mathbf{v}_i, T) \xrightarrow{\delta_i} (q'_i, \mathbf{v}'_i, T)$ is a discrete step of some[2] automaton $\mathcal{A}_i$. A global time step is

$$((q_1, \ldots, q_n), (\mathbf{v}_1, \ldots, \mathbf{v}_n), T) \xrightarrow{t} ((q_1, \ldots, q_n), (\mathbf{v}_1 + t\mathbf{1}, \ldots, \mathbf{v}_n + t\mathbf{1}), T + t)$$

such that for every q_i, S_{q_i} is satisfied by $\mathbf{v}_i + t\mathbf{1}$ and by $(q_1, \ldots, q_n)$. This means that whenever one automaton makes a discrete transition at time T, any other automaton

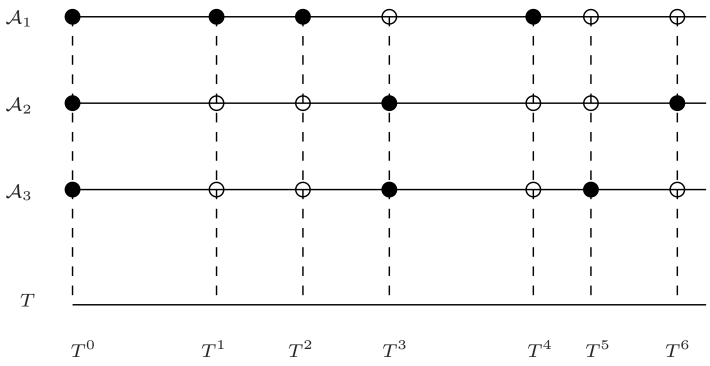

Fig. 1. A fragments of a run of 3 automata where the black dots represent discrete transitions. The local minimal runs need to be refined by splitting time steps in order to to conform with the minimal run of the global automaton. For example, $\mathcal{A}_1$ splits its time step at $[T^2, T^4)$ into two steps at $[T^2, T^3)$ and $[T^3, T^4)$.

that makes a time step in an interval $[T', T'')$ containing T, must "split" that step (see Figure 1).

To build the formulae we replace, as before, clock variables by date variables $\mathcal{X}_i$ for each automaton ($\mathcal{X} = \bigcup \mathcal{X}_i$) and use one global time stamp variable T common to all automata. We construct for each $\mathcal{A}_i$ its step formulae $\Phi_i(\mathcal{B}, \mathcal{X}_i, T, \mathcal{B}'_i, \mathcal{X}'_i, T')$ and the conjunction of those characterizes a global step. The use of the same variable T by all automata guarantees synchronization, i.e. that whenever $\mathcal{A}_i$ refers to the state variables of $\mathcal{A}_j$, their values correspond to the same time instant.

When the automata are loosely-coupled it might be more economical to express the steps of each automaton using its *private time-scale* and impose synchronization between those time scales only when the automata interact (this approach was introduced in [BJLY98] as an attempt to apply partial-order methods to timed automata). For each automaton $\mathcal{A}_i$ we use a different time-stamp variable T_i ($\mathcal{T} = \bigcup T_i$) such that the values of its date variables are $\mathcal{X}_i = T_i - \mathcal{C}_i$. Then we have, for every $\mathcal{A}_i$ and a reset ρ_i

$$\Phi_{\rho_i}(\mathcal{X}_i, \mathcal{X}'_i, T_i) \equiv \bigwedge_{C_m \in \rho_i} X'_m = T_i \wedge \bigwedge_{C_m \notin \rho_i} X'_m = X_m.$$

Time passage for $\mathcal{A}_i$ is written as

$$\Phi_{\tau_i}(\mathcal{X}_i, T_i, \mathcal{X}'_i, T'_i) \equiv \bigwedge_m X'_m = X_m \wedge T'_i - T_i \geq 0.$$

The transition formula for $\delta_i \in \Delta_i$ is

$$\Psi_{\delta_i}(\mathcal{B}, \mathcal{X}, \mathcal{T}, \mathcal{B}'_i, \mathcal{X}', \mathcal{T}') \equiv \begin{array}{l} \Phi_{q_i}(\mathcal{B}_i) \wedge \Phi_{g_i}(\mathcal{B}, \mathcal{X}_i, T_i) \wedge \Phi_{\rho_i}(\mathcal{X}_i, \mathcal{X}'_i, T_i) \wedge \\ \Phi_{q'_i}(\mathcal{B}'_i) \wedge T_i = T'_i \wedge \bigwedge_{j \in J_{\delta_i} \cup J'_i} T_i = T_j \end{array}$$

[2] This definition corresponds to full interleaving of transitions but in the implementation we allow several simultaneous transitions in the same step.

where $\Phi_{g_i}(\mathcal{B}, \mathcal{X}_i, T_i)$ is the substitution of $T_i - \mathcal{X}_i$ and the corresponding values of the state-variables of the other automata in g_i. Note that the last condition enforces identity on time stamps only for automata whose state is important for the transition, i.e. those that guard it and those whose time steps might be disabled due to the transition. Needless to say, only the state variables of the automata in J_{δ_i} will appear in the formula. The formula for a time step at q_i is

$$\Psi_{q_i}(\mathcal{B}, \mathcal{X}, \mathcal{T}, \mathcal{B}', \mathcal{X}', \mathcal{T}') \equiv \begin{array}{l} \Phi_{q_i}(\mathcal{B}_i) \wedge \Phi_{\tau_i}(\mathcal{X}_i, T_i, \mathcal{X}_i', T_i') \wedge \Phi_{s,q_i}(\mathcal{B}, \mathcal{X}', T_i') \wedge \\ \Phi_{q_i}(\mathcal{B}_i') \wedge \bigwedge_{j \in J_{q_i}} (T_i = T_j) \end{array}$$

where $\Phi_{s,q_i}(\mathcal{B}, \mathcal{X}_i', T_i')$ is just the substitution of $T_i' - \mathcal{X}_i'$ and all the state variables in S_{q_i}. The formula for a step of $\mathcal{A}_i$ is

$$\Psi_i(\mathcal{B}, \mathcal{X}_i, \mathcal{T}, \mathcal{B}_i', \mathcal{X}_i', \mathcal{T}') \equiv \bigvee_{q \in Q_i} \Psi_q \vee \bigvee_{\delta \in \Delta_i} \Psi_\delta.$$

The formula for a global step is

$$\Psi(\mathcal{B}, \mathcal{X}, \mathcal{T}, \mathcal{B}', \mathcal{X}', \mathcal{T}') \equiv \bigwedge_{i \in I} \Psi_i(\mathcal{B}, \mathcal{X}_i, \mathcal{T}, \mathcal{B}_i, \mathcal{X}_i', \mathcal{T}')$$

and the formula for a run with at most k transitions is

$$\Psi_k \equiv \Psi(\mathcal{B}^0, \mathcal{X}^0, \mathcal{T}^0, \mathcal{B}^1, \mathcal{X}^1, \mathcal{T}^1) \wedge \ldots \wedge \Psi(\mathcal{B}^{k-1}, \mathcal{X}^{k-1}, \mathcal{T}^{k-1}, \mathcal{B}^k, \mathcal{X}^k, \mathcal{T}^k).$$

In this construction we pay the price of introducing more time-stamp variables in order to obtain less steps for the same run.

5 The Solver

5.1 Handling Mixed Constraints

The challenge for a satisfiability checker for DL lies, of course, in the interaction between Boolean and numerical constraints. We sketch some of the popular approaches to this problem. Given the scope of the topic and its inter-disciplinary nature, this is not intended to be an exhaustive nor an objective survey.

Numerical constraints were traditionally treated in the context of continuous optimization where efficient algorithms exist, for example, for conjunction of linear constraints (linear programming). Attempting to extend this approach to mixed constraints leads to the so-called mixed integer-linear programming where Booleans are viewed as integers. When there are only few such "integer" variables one may convert them into reals ranging in $[0, 1]$, use efficient linear programming algorithms to find a satisfying assignment for the relaxed problem and then try to push the assignments for the converted Booleans back to $\{0, 1\}$. Although certain classes of problems admit successful relaxation schemes, we do not believe that this approach will work when the combinatorial part of the problem is significant and the discrete variables have no metric meaning.

An opposite approach, which tries to keep everything inside the propositional world, is to replace every numerical constraint by a new Boolean variable (similarly to what we partially do while transforming to CNF) until a purely propositional problem remains. After a satisfying assignment for this problem is found, linear programming can be used to see whether the numerical constraints implied by the extended assignment are satisfiable. If they are not satisfiable, a new propositional instance should be found, and so on. The advantage of this approach is in the ability to use of-the-shelf efficient SAT solvers. However, the interaction between the numerical and logical constraints is restricted only to the final phase and this might lead to a lot of backtracking.

The approach, often described under the title of *constraint propagation* or *constraint logic programming* [JM94], treats both types of constraints rather equally. Here the key idea is an incremental search in the space of solution, where every decision concerning one variable can be propagated to reduce the domains of other variables. Such methods have been applied rather successfully to problems with increasingly complex numerical constraints, both linear and non-linear. Some recent "hybrid" techniques that combine constraint propagation with linear programming proved to be very powerful. More on the relation between these approaches can be found in [H00].

Given that DL, with its restricted class of linear constraints, is perhaps the most conservative numerical extension of propositional logic, we have chosen to construct our solver as an extended propositional solver. Our algorithm gives priority to propositional reasoning but derives information from numerical constraints as soon as they become isolated and easy to manipulate. The key idea of our approach is that a conjunction of difference constraints can be represented using a *difference bounds matrix*, a data-structure used extensively in the verification of timed automata [Dil89]. The Floyd-Warshall algorithm is used to normalize the constraints, to find contradictions in the set of numerical clauses and also to extract a solution from a consistent set of inequalities in polynomial time.

Typical SAT solvers perform both syntactic transformations of formulae that may simplify clauses and remove variables, and a search in the space of valuations for the remaining variables. Our solver, inspired by the Davis-Putnam procedure, gives priority to the former and resorts to search ("branching") only when simplifications are impossible. In other popular solvers such as GRASP [MS99] and Chaff [MMZ$^+$], search is the primary mechanism and syntactic transformations are incorporated as "learning", that is adding new clauses implied by failures in the search.

5.2 Boolean Proof Methods

In this section we sketch the way we treat the Boolean part, some of which is common to other approaches for realizing the Davis-Putnam proof procedure (see, e.g. [Zha95]). This will serve as an introduction to the treatment of numerical constraints (there are some interesting analogies between the Boolean and numerical proof methods). Our presentation largely follows that of [GW99].

The proof method relies on applying satisfiability-preserving transformations to 3CNF DL formulae, some of which might reduce the number of satisfying valuations. Successive applications of these rules can lead to a satisfying valuation or to a (resolution) proof of unsatisfiability.

Simplifications: Elimination of redundant clauses which are weaker than others, independently of the rest of the formula. On the level of literals, a literal L_2 is weaker than another literal L_1 if for any valuation v, $v \models L_1$ implies $v \models L_2$. For Boolean literals this is the case for identical literals. For numerical constraints, $X - Y \geq c$ is weaker than $X - Y \geq d$ if $d \geq c$. This weaker-than relation generalizes to clauses in the obvious way.

Elimination of redundant literals by resolution: If there are two clauses $C_1 = L_1 \vee C'$ and $C_2 = \neg L_1 \vee C'$, replace C_1 and C_2 by C'.

Boolean unit resolution: If there is a clause L_1 and a clause $C = L_2 \vee C'$ such that L_1 and L_2 are contradictory, replace C by C'. Boolean literals are contradictory if one is the negation of the other.

Boolean pure variables detection: If there is a boolean literal L such that the formula can be writtem as $\bigwedge_i C_i \wedge \bigwedge_j (L \vee C'_j)$ where L does not occur in any of the clauses C_i and C'_j. Then we can set $L := T$ and the formula is reduced $\bigwedge_i C_i$.

Detection of logical cycles: If there is a chain of two literal clauses $\neg L_1 \vee L_2, \neg L_2 \vee L_3, \ldots \neg L_n \vee L_1$, apply a renaming of $L_2, \ldots, L_n$ to L_1 (this renaming has to be remembered in order to reconstruct complete valuations of the original formula). A cycle including the same literal negatively and positively exposes a contradiction and the formula is unsatisfiable. In practice, the detection of cycles is implemented using Tarjan's linear-time algorithm for computing strongly connected components [Tar72]. For performance considerations, this rule is only applied if the number of two literal clauses is below a certain value (800 in the current implementation).

Detection of Boolean contradictions: An empty clause is not satisfiable, hence the entire formula containing an empty clause is not satisfiable. Note that numerical contradictions will show up as Boolean contradictions via resolution.

The empty conjunction: If we arrive at a formula containing no more clauses, it is trivially satisfiable by any valuation.

All of the above transformations can reduce the size of a formula without affecting its semantics. Sometimes, however, checking the conditions for applying these rules might be costly (quadratic in the size of the formula).

The Davis-Putnam rules go beyond these transformations by replacing a formula by a stronger one while preserving (non)satisfiability. Instead of describing the general Davis-Putnam framework, we only mention an interesting rule which preserves the 3CNF structure without introducing new variables.

Boolean Davis-Putnam rule: Supposing that there is a Boolean literal L such that the formula can be represented as $\bigwedge_i C_i \wedge \bigwedge_j (L \vee C'_j) \wedge \bigwedge_k (\neg L \vee L_k)$ where L does not occur in any of the clauses C_i, C'_j. That is, all negative occurrences of L are in 2-clauses. Then we can set $L := \bigwedge_k L_k$ (and memorize this assignment) and substitute $\bigwedge_k L_k$ for L throughout the formula.

Note that this transformation, while preserving satisfiability, can decrease the number of solutions. Secondly, while going back to 3CNF, some clauses may have to be copied in case of more than a single negative occurrence of L. It is a question of heuristics, when the application of this rule is sensible. Obviously, in the case of a single negative occurrence of L this will result in a simplification.

Branching When the above transformations yield neither a solution nor a contradiction we resort to branching over valuations of Boolean variables: A Boolean variable is chosen and either assumed to be false or true, thus eliminating it. By recursion, eventually all variables will be eliminated. However, the scheme as stated requires backtracking, possibly leading to an exponential-time linear-space complexity.

5.3 Numerical Proof Methods

The main novelty of our approach is in the treatment of numerical difference constraints. We sketch here the major ideas as applicable to the integer interpretation.

Numerical unit resolution When a formula contains a numerical 1-literal clause $X - Y \geq c$, we can replace numerical literals $X - Y \geq d$ with $d \leq c$ by T and numerical literals $Y - X \geq d$ with $c > -d$ by F.

Computing numerical implications: Consider a chain of clauses $X_1 - X_2 \geq c_1, \ldots,$ $X_{n-1} - X_n \geq c_n$. For every j, k such that $1 \leq j < k \leq n$ we can conclude that $X_j - X_k \geq \sum_{j \leq i \leq k} c_i$. These implied inequalities can be used to remove weaker numerical literals from the formula. In particular, when the chain involves a cycle (X_j and X_k are identical) then it can be replaced by F if it is positive. Moreover, if this sum is zero, one can eliminate the variables $X_{j+1}, \ldots, X_{k-1}$ and express them using X_j.

Similarly to the treatment of Boolean chains and cycles, the above numerical implications are computed using a graph algorithm, in this case the Floyd-Warshall *all vertex shortest path algorithm with negative weights* [CLRS01]. The data-structure for representing conjunctions of difference constraints is the *difference bounds matrix* used extensively in the verification of timed automata. In this matrix an inequality of the form $X_i - X_j \geq c$ is represented by putting $-c$ in the (i, j)-entry of the matrix (when two variables are not related by a constraint c is set to $-\infty$).

In the following, we develop an analogue of the Davis-Putnam rule for numerical constraints. We say that in a numerical literal $X - Y \geq c$ the variable X occurs positively and Y occurs negatively. Let v and v' be two valuation which are identical except for one variable X. One can see that if $v'(X) \geq v(X)$ then $v \models L$ implies $v' \models L$ for every literal L where X occurs positively and $v' \models L$ implies $v \models L$ when X occurs negatively in L.

Davis-Putnam for literal $\geq$-constraints: Assume, we have only a few (ideally, one) positive occurrences of X in literal clauses $X - Y_i \geq c_i$ with $i \in I$ and all other occurrences – in particular those in mixed clauses – are negative. Then we can set $X := max\{Y_i + c_i \mid i \in I\}$ (or $X := Y_i + c_i$ in the ideal case) and eliminate X from all clauses: Each literal $Z - X \geq d$ will be replaced by a conjunction $\bigwedge_{i \in I} Z - Y_i \geq d + c_i$. Analogously, if we have only a few negative occurrence of X in the one-literal clauses $Y_i - X \geq c$ with $i \in I$, we can safely set $X := min\{Y - c_i \mid i \in I\}$.

This rule is correct in not affecting the satisfiability of the formula because any valuation v satisfying it must satisfy $v(X) \geq v(Y_i) + c_i$. The valuation $v' = v[X :=$

$max\{v(Y_i) + c_i\}]$ where only the value of X is changed such that $v'(X) \leq v(X)$ also satisfies the formula, because it satisfies $X - Y_i \geq c_i$ for all $i \in I$ and all other clauses containing negative occurrences of X will also remain true.

5.4 The Overall Algorithm.

The algorithm consists of two major parts: The branching procedure and the non-branching rules. Since the cost of Davis-Putnam rules is potentially higher than that of the other rules, Boolean and numerical rules are performed in the following order:

1. Boolean simplifications;
2. Unit resolution;
3. Strongly connected component algorithm on the two literal Boolean clauses with subsequent detection of contradictions or elimination of equivalent literals;
4. Shortest path algorithm for the normalization of numerical constrains with subsequent detection of contradictions or elimination of tightly coupled variables and numerical unit resolution;
5. Application of the Boolean Davis-Putnam rule;
6. Application of the numerical Davis-Putnam rule;
7. Pass to branching.

Whenever a change in the clause set occurs at one of the stages (1)-(6), the algorithm restarts at (1).

6 Implementation and Experimental Results

The solver has been written from scratch in order to keep its size small. In its current status it still needs a lot of tuning and application-specific heuristics, in particular, for selecting variables for branching. Consequently, its current performance is inferior to that of more mature tools for timed verification. We hope that this situation will change in the near future. We will report the result of preliminary experiments after a brief description of the implementation architecure.

6.1 Implementation

The central data-structure of the solver is the clause table in which we keep Boolean and mixed clauses, while while purely numerical clauses are stored in a difference bounds matrix. The interaction between the two happens when, following a simplification, a mixed clause is transformed into a numerical literal and passes from the clause table to the matrix. Numerical implications are computed on the matrix and may affect Boolean clauses, for example by numerical unit resolution.

Due to branching we need to organize these data structures in a stack. Heuristic improvements can be used to render stack operations less expensive, e.g. a special technique for storing stacks of sparse matrices.

We have tried to keep the translation from DL to 3CNF separated as much as possible from the rest of the solver. And indeed, a new version of the solver, not restricted to 3CNF has been written recently in a very short time. This version consumes less memory but still needs more tuning to compete in time performance with the 3CNF version.

We have started implementing translators from various formats of timed automata as used in the tools Kronos, OpenKronos and IF [Y97,BDM+98,BFG+00]. Since these tools admit a variety of syntax, synchronization mechanisms and semantic definitions, this process is not yet complete. The translations that are currently working are a direct translation from job-shop scheduling problems[3], a translation from flat timed automata in the Kronos format, and a global-time compositional translation for the sub-class of timed automata corresponding to digital circuits with bi-bounded inertial delays [BS94] following their modeling as timed automata [MP95,BJMY].

6.2 Experimental Results

Long Runs of Simple Automata As a first example we took a simple timed automaton with 4 states and one clock and created, via our translator, DL formulae for paths of varying length. Table 1 show how the size and properties of the formulae change with the length of the path, as well as the time it takes to check satisfiability. The good results here are misleading, because most of the complexity in this example is along the temporal dimension and hence the solver does not need to perform a lot of branching.

path length	# bool vars	# real vars	# bool clauses	# mixed clauses	time (secs)
1	51	7	87	52	0.01
10	492	34	888	520	0.24
20	982	64	1778	1040	1.46
50	2452	154	4448	2600	10.34
100	4902	304	8898	5200	43.20

Table 1. Size of reachability formulae (in 3CNF) and solution time as a function of path length for a simple automaton. In all cases the formula is satisfiable.

Job-Shop Scheduling The classical job-shop scheduling problem translates very naturally into DL. The problem can be stated as an optimization problem ("find an optimal schedule") or as a decision problem ("is there a schedule of length smaller than L?"). We have experimented with a well-known benchmark example taken from [ABZ88] with 10 machines and 10 jobs, each having 10 steps. The known optimal schedule is of length 1179. The 3CNF DL formula for a feasible schedule has 1452 Boolean variables, 102 real variables, 1901 Boolean clauses and 2453 mixed clauses. While posing the verification problem we observe a phenomenon which, we think, is very typical in

[3] This translation does not go through timed automata but encodes the obvious constraints on the start times of steps in jobs.

such situations: when L is much larger than the length of the optimal schedule, a satisfying assignment is easily found. Likewise, when L is much smaller than the optimum, we detect quickly a contradiction. As we approach the optimum from both sides, the computational cost grows. The results appear in Table 2.

L	answer	time(secs)
100	No	4.72
500	No	4.47
750	No	4.57
1000	No	31.03
1100	No	∞
1179	Yes	∞
1200	Yes	∞
1300	Yes	∞
1400	Yes	18.88
1500	Yes	13.36
2000	Yes	12.59
3000	Yes	12.79
5000	Yes	13.59
10000	Yes	14.53

Table 2. The results for the job-shop benchmark. The second column indicates the (a-priori known) answer to the question whether there exists a schedule of length $\leq L$. The symbol ∞ indicates that the solver did not terminate within 25 minutes.

All experiments were performed on a standard PC powered by a 600MHz Pentium III. The maximum memory usage reached was about 90MB (for the hard job-shop examples and the formula for a path of length 100 for the simple automaton). Other experiments required less than 10MB.

7 Related Work and Conclusions

The idea to extend the applicability of SAT-based verification from finite-state systems to systems augmented with unbounded variables or clocks is a very natural one and has been pursued independently by several groups. In [ACG99] a solver for a similar logic was applied to AI temporal planning problems. The closest work to ours is that of [ABC$^+$02,ACKS02] where the authors develop an extended SAT solver to verify timed automata against temporal logic specifications. The major differences with respect to the present paper is in their use of a more general linear constraint solver, and in the limited interaction between the Boolean and numerical parts. The work of [MRS02] considers more general numerical constraints and gives priority to propositional reasoning by encoding numerical constraints as Booleans and then using decision procedures to eliminate "spurious solutions". Other investigations in this methodology appear in [SSB02] where DL formulae are called separation formulae.

We have proposed an alternative approach for solving timing related verification and optimization problems. It is worth mentioning other works, such as [BFH+01,AM01], that go in the opposite direction by applying timed automata verification techniques to scheduling problems that were traditionally solved using constraint resolution. We hope that all this effort will eventually lead to performance improvements and to a better understanding of the computational difficulty of temporal reasoning.

References

[AM01] Y. Abdeddaïm and O. Maler, Job-Shop Scheduling using Timed Automata *Proc. CAV'01*, 478-492, LNCS 2102, Springer 2001.

[ABZ88] J. Adams, E. Balas and D. Zawack, The Shifting Bottleneck Procedure for Job Shop Scheduling, *Management Science* 34, 391-401, 1988.

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183-235, 1994.

[ACG99] A. Armando, C. Castellini and E. Giunchiglia, SAT-based Procedures for Temporal Reasoning, *Proc. ECP'99*, LNCS, Springer, 1999.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse, Data Structures for the Verification of Timed Automata, *Proc. Hybrid and Real-Time Systems*, 346-360, LNCS 1201, Springer, 1997.

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics and R. Sebastiani, A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions, in *Proc. CADE'02*, 193-208, LNCS 2392, Springer, 2002.

[ACKS02] G. Audemard, A. Cimatti, A. Kornilowics and R. Sebastiani, Bounded Model Checking for Timed Systems, Technical report ITC-0201-05, IRST, Trento, 2002.

[BFH+01] G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson and J. Romijn, Efficient Guiding Towards Cost-Optimality in UPPAAL, *Proc. TACAS 2001*, 174-188, LNCS 2031, Springer, 2001.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius and W. Yi, Partial Order Reductions for Timed Systems, *Proc. Concur'98*, 485-500, LNCS 1466, Springer, 1998.

[BCRZ99] A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu, Verifying Safety Properties of a PowerPC Microprocessor using Symbolic Model Checking without BDDs, *Proc. CAV'99*, 60-71, LNCS 1633, Springer, 1999.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu, Symbolic Model Checking without BDDs, *Proc. TACAS'99*, 193-207, LNCS 1579, Springer, 1999.

[BM00] O. Bournez and O. Maler, On the Representation of Timed Polyhedra, *Proc. ICALP 2000*, 793-807, LNCS 1853, Springer, 2000.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, Kronos: a Model-Checking Tool for Real-Time Systems, *Proc. CAV'98*, LNCS 1427, Springer, 1998.

[BFG+00] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-.P. Krimm and L. Mounier, IF: A Validation Environment for Timed Asynchronous Systems, *Proc. CAV'00*, LNCS, Springer, 2000.

[BJMY] M. Bozga, H. Jianmin, O. Maler and S. Yovine, Verification of Asynchronous Circuits using Timed Automata, *Proc. TPTS'02*, 2002.

[Bry86] R.E. Bryant, Graph-based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* 35, 677-691, 1986.

[BS94] J.A. Brzozowski and C-J.H. Seger, *Asynchronous Circuits*, Springer, 1994.

[BCM+93] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, Symbolic Model-Checking: 10^{20} States and Beyond, *Proc. LICS'90*, IEEE, 1990.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 2001.

[Dil89] D.L. Dill, Timing Assumptions and Verification of Finite-State Concurrent Systems, *Proc. CAV'89*, 197-212, LNCS 407, Springer, 1989.

[EC82] E.A. Emerson and E.M. Clarke, Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming* 2, 241-266, 1982.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[GW99] J.F. Groote and J.P. Warners. The Propositional Formula Checker HeerHugo. Technical Report 691, Centrum voor Wiskunde en Informatica (CWI) Amsterdam, 1999.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic Model-checking for Real-time Systems, *Information and Computation* 111, 193-244, 1994.

[H00] J. Hooker, *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, Wiley, 2000

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey, *Journal of Logic Programming*, 19/20, 503-581, 1994.

[LPWY99] K. Larsen, J. Pearson, C. Weise, and W. Yi, Clock difference diagrams. *Nordic Journal of Computing* 6, 271-298, 1999.

[LP84] O. Lichtenstein, A. Pnueli, Checking that Finite-state Concurrent Programs Satisfy their Linear Specification, *Proc. POPL'84*, 97-107, ACM, 1984.

[MP95] O. Maler and A. Pnueli, Timing Analysis of Asynchronous Circuits using Timed Automata, *Proc. CHARME'95*, 189-205, LNCS 987, Springer, 1995.

[MS99] J.P. Marques-Silva and K.A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers* 48, 506-21, 1999.

[McM93] K.L. McMillan, *Symbolic Model-Checking: an Approach to the State-Explosion problem*, Kluwer, 1993.

[MLAH99] J. Møller, J. Lichtenberg, H. Andersen, and H. Hulgaard, Difference Decision Diagrams, *Proc. CSL'99*, 1999.

[MMZ+] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an Efficient SAT Solver, *Proc. DAC 2001*, 2001.

[MRS02] L. de Moura, H. Rueß and M. Sorea, Lazy Theorem Proving for Bounded Model Checking over Infinite Domains, *Proc. CADE'02*, 437-453, LNCS 2392, Springer, 2002.

[QS81] J.P. Queille and J. Sifakis, Specification and Verification of Concurrent Systems in Cesar, *Proc. 5th Int. Symp. on Programming*, 337-351, LNCS 137, Springer, 1981.

[SS90] G. Stålmarck and M. Saflund, Modeling and Verifying Systems and Software in Propositional Logic, *Safety of Computer Control Systems (SAFECOMP'90)*, 31-36, Pergamon Press, 1990.

[SSB02] O. Strichman, S.A. Seshia, and R.E. Bryant, Deciding Separation Formulas with SAT, in *Proc. CAV'2002*, Springer, 2002.

[Tar72] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 146-160, 1972.

[T70] G. Tseitin, On the Complexity of Derivation in Propositional Calculus, in *Studies in Constructive Mathematics and Mathematical Logic* 2, 115-125, Consultants Bureau, New York, 1970.

[VW86] M.Y. Vardi and P. Wolper, An Automata-theoretic Approach to Automatic Program Verification, *Proc. LICS'86*, 322-331, IEEE, 1986.

[Y97] S. Yovine, Kronos: A Verification Tool for Real-time Systems, *International Journal of Software Tools for Technology Transfer* 1, 123-133, 1997.

[Zha95] G. Zhang. The Davis-Putnam Resolution Procedure, In *Advances in Logic Programming and Automated Reasoning*, volume 2. Ablex Publishing Corporation, 1995.