Efficient On-the-Fly Algorithms for the Analysis of Timed Games

Franck Cassez[1,⋆], Alexandre David[2], Emmanuel Fleury[2], Kim G. Larsen[2], and Didier Lime[2]

[1] IRCCyN, UMR 6597, CNRS, France
Franck.Cassez@irccyn.ec-nantes.fr
[2] Computer Science Department,
CISS (Center for Embedded Software Systems), Aalborg University, Denmark
{adavid, fleury, kgl, didier}@cs.aau.dk

Abstract. In this paper, we propose the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties

The algorithm we propose is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka [15] for linear-time model-checking of finite-state systems. Being on-the-fly, the symbolic algorithm may terminate long before having explored the entire state-space. Also the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure.

Various optimizations of the basic symbolic algorithm are proposed as well as methods for obtaining time-optimal winning strategies (for reachability games). Extensive evaluation of an experimental implementation of the algorithm yields very encouraging performance results.

1 Introduction

On-the-fly algorithms offer the benefit of settling properties of individual system states (*e.g.* an initial state) in a local fashion and without necessarily having to generate or examine the entire state-space of the given model. For finite-state (untimed) systems the search for optimal (linear) on-the-fly or local algorithms has been a very active research topic since the end of the 80's [12,4,15] and is one of the most important techniques applied in finite-state model-checkers using enumerative or explicit state-space representation, as is the case with SPIN [10], which performs on-the-fly model-checking of LTL properties.

Also for timed systems, on-the-fly algorithms have been absolutely crucial to the success of model-checking tools such as Kronos [8] and Uppaal [13] in their analysis of timed automata based models [2]. Both reachability, safety as well as general liveness properties of such timed models may be decided using on-the-fly algorithms exploring the reachable state-space in a (symbolic) forward manner with the possibility of early termination. More recently, timed automata technology has been successfully applied to optimal scheduling problems with

⋆ Work supported by ACI Cortos, a program of the French government. Visits to Aalborg supported by CISS, Aalborg University, Denmark.

guiding and pruning heuristics being added to yield on-the-fly algorithms which quickly lead to near-optimal (time- or cost-wise) schedules [5,3,11,18].

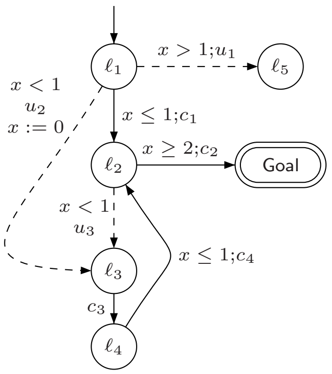


Fig. 1. A Timed Game Automaton

We consider timed game automata and how to decide the existence of a winning strategy *w.r.t.* reachability or safety. As an example, consider the timed game automaton A of Fig. 1 consisting of a timed automaton with one clock x and two types of edges: controllable (c_i) and uncontrollable (u_i). The reachability game consists in finding a strategy for a controller, *i.e.* when to take the controllable transitions that will guarantee that the system, regardless of when and if the opponent chooses to take uncontrollable transitions, will eventually end up in the location Goal. Obviously, for all initial states of the form (ℓ_1, x) with $x \leq 1$ there is such a winning strategy[3].

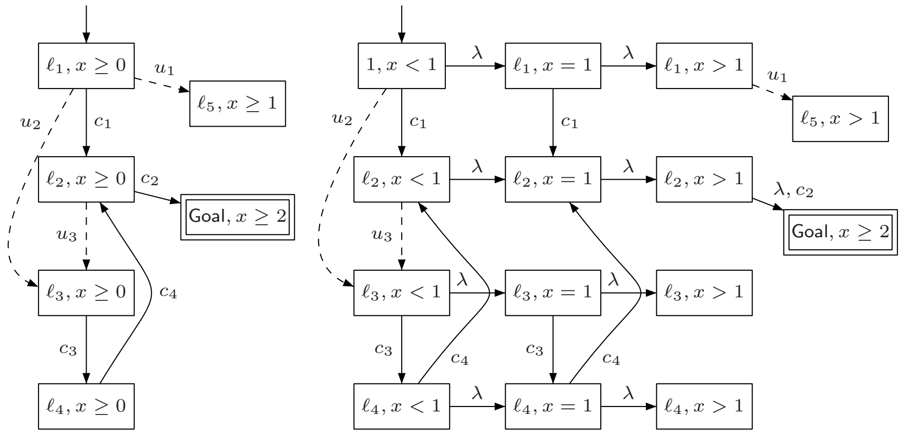
Though such timed game automata for long have been known to be decidable [16,6,9] there is still a lack of efficient and truly on-the-fly algorithms for their analysis. Most of the suggested algorithms are based on backwards fix-point computations of the set of winning states [16,6,9]. In contrast, the on-the-fly algorithms used for model-checking timed automata models (*w.r.t.* reachability) make a forward symbolic state-space exploration resulting in the so-called *simulation graph*. However, the simulation graph is by itself too abstract to be used as the basis for an on-the-fly algorithm for computing winning strategies. Fig. 2 (a) gives the simulation graph of the timed game automata of Fig. 1, which incorrectly classifies the initial state as being uncontrollable when viewed as a finite-state game.

As a remedy to this problem, the authors of [20,1] propose a partially on-the-fly method for solving reachability games for a timed game automaton A. However, this method involves an extremely expensive preprocessing step in which the quotient graph of the dense time transition system S_A *w.r.t.* time-abstracted bisimulation[4] needs to be built. Once obtained this quotient graph may be used with any on-the-fly game-solving algorithm for untimed (finite-state) systems. As an illustration, Fig. 2 (b) gives the time abstracted quotient graph for the timed game automaton of Fig. 1. It should be easy for the reader to see that the initial state will now (correctly) be classified as controllable.

In this paper, we propose an efficient, truly on-the-fly algorithm for the computation of winning states for timed game automata. Our algorithm is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka [15] for

[3] A winning strategy would consist in taking c_1 immediately in all states (ℓ_1, x) with $x \leq 1$; taking c_2 immediately in all states (ℓ_2, x) with $x \geq 2$; taking c_3 immediately in all state (ℓ_3, x) and delaying in all states (ℓ_4, x) with $x < 1$ until the value of x is 1 at which point the edge c_4 is taken.

[4] A time-abstracted bisimulation is a binary relation on states preserving discrete states and abstracted delay-transitions.



(a) Simulation Graph (b) Time-abstracted Quotient Graph

Fig. 2. Simulation and time-abstracted quotient graph of Fig. 1 (λ is for time elapsing)

linear-time model-checking of finite-state systems. Being on-the-fly, the symbolic algorithm may terminate before having explored the entire state-space, *i.e.* as soon as a winning strategy has been identified. Also the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure.

The rest of the paper is organized as follows. Section 2 provides definitions and preliminaries about timed game automata and the classic backwards algorithm for solving them. Section 3 presents our instantiation of the general on-the-fly algorithm of Liu & Smolka [15] to untimed reachability games. Then, in Section 4, we present our symbolic extension of this algorithm, providing a first forward, zone-based and fully on-the-fly algorithm for solving timed reachability games. Section 5 discusses few optimizations of the basic algorithm and how to apply the algorithm to determine time-optimal winning strategies. Section 6 presents experimental evaluation of an efficient implementation of the algorithm to determine (time-optimal) winning strategies. The performance results obtained are very encouraging. Finally, Section 7 presents conclusion and future work.

2 Backward Algorithms for Solving Timed Games

Timed Game Automata [16] (TGA) were introduced for control problems on timed systems. This section recalls basic results of controller synthesis for TGA.

Let X be a finite set of real-valued variables called clocks. We note $\mathcal{C}(X)$ the set of constraints φ generated by the grammar: $\varphi ::= x \sim k \mid x - y \sim k \mid \varphi \wedge \varphi$ where $k \in \mathbb{Z}$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X)$ is the subset of $\mathcal{C}(X)$ that uses only rectangular constraints of the form $x \sim k$. A *valuation* of the variables in X is a mapping $X \mapsto \mathbb{R}_{\geq 0}$ (thus $\mathbb{R}_{\geq 0}^{X}$). We write $\mathbf{0}$ for the valuation that assigns 0 to each clock. For $Y \subseteq X$, we denote by $v[Y]$ the valuation assigning 0 (*resp.* $v(x)$) for any $x \in Y$ (*resp.* $x \in X \setminus Y$). We denote $v + \delta$ for $\delta \in \mathbb{R}_{\geq 0}$

the valuation *s.t.* for all $x \in X$, $(v + \delta)(x) = v(x) + \delta$. For $g \in \mathcal{C}(X)$ and $v \in \mathbb{R}^X_{\geq 0}$, we write $v \models g$ if v satisfies g and $[\![g]\!]$ denotes the set of valuations $\{v \in \mathbb{R}^X_{\geq 0} \mid v \models g\}$. A *zone* Z is a subset of $\mathbb{R}^X_{\geq 0}$ *s.t.* $[\![g]\!]= Z$ for some $g \in \mathcal{C}(X)$.

2.1 Timed Game Automata and Simulation Graph

Definition 1 (Timed Automaton [2]). *A* Timed Automaton *(TA) is a tuple* $A = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ *where* L *is a finite set of* locations*,* $\ell_0 \in L$ *is the* initial *location,* Act *is the set of* actions*,* X *is a finite set of* real-valued clocks*,* $E \subseteq L \times \mathcal{B}(X) \times \mathsf{Act} \times 2^X \times L$ *is a finite set of* transitions*,* $\mathsf{Inv} : L \to \mathcal{B}(X)$ *associates to each location its* invariant*.*

A *state* of a TA is a pair $(\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ that consists of a discrete part and a valuation of the clocks. From a state $(\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ *s.t.* $v \models \mathsf{Inv}(\ell)$, a TA can either let time progress or do a discrete transition and reach a new state. This is defined by the transition relation $\longrightarrow$ built as follows: for $a \in \mathsf{Act}$, $(\ell, v) \xrightarrow{a} (\ell', v')$ if there exists a transition $\ell \xrightarrow{g,a,Y} \ell'$ in E *s.t.* $v \models g$, $v' = v[Y]$ and $v' \models \mathsf{Inv}(\ell')$; for $\delta \geq 0$, $(\ell, v) \xrightarrow{\delta} (\ell, v')$ if $v' = v + \delta$ and $v, v' \in[\![\mathsf{Inv}(\ell)]\!]$. Thus the semantics of a TA is the labeled transition system $S_A = (Q, q_0, \longrightarrow)$ where $Q = L \times \mathbb{R}^X_{\geq 0}$, $q_0 = (\ell_0, \mathbf{0})$ and the set of labels is $\mathsf{Act} \cup \mathbb{R}_{\geq 0}$. A *run* of a timed automaton A is a sequence of alternating time and discrete transitions in S_A. We use $\mathsf{Runs}((\ell, v), A)$ for the set of runs that start in (ℓ, v). We write $\mathsf{Runs}(A)$ for $\mathsf{Runs}((\ell_0, \mathbf{0}), A)$. If ρ is a finite run we denote $last(\rho)$ the last state of the run and $\mathsf{Duration}(\rho)$ the total elapsed time all along the run.

The analysis of TA is based on the exploration of a finite graph, the *simulation graph*, where the nodes are *symbolic states*; a symbolic state is a pair (ℓ, Z) where $\ell \in L$ and Z is a zone of $\mathbb{R}^X_{\geq 0}$. Let $X \subseteq Q$ and $a \in \mathsf{Act}$ we define the a-successor of X by $\mathsf{Post}_a(X) = \{(\ell', v') \mid \exists (\ell, v) \in X, (\ell, v) \xrightarrow{a} (\ell', v')\}$ and the a-predecessor $\mathsf{Pred}_a(X) = \{(\ell, v) \mid \exists (\ell', v') \in X, (\ell, v) \xrightarrow{a} (\ell', v')\}$. The timed successors and predecessors of X are respectively defined by $X^{\nearrow} = \{(\ell, v + d) \mid (\ell, v) \in X \cap [\![\mathsf{Inv}(\ell)]\!], (\ell, v + d) \in[\![\mathsf{Inv}(\ell)]\!],\ d \in \mathbb{R}_{\geq 0}\}$ and $X^{\swarrow} = \{(\ell, v - d) \mid (\ell, v) \in X,\ d \in \mathbb{R}_{\geq 0}\}$. Let $\to$ be the relation defined on symbolic states by: $(\ell, Z) \xrightarrow{a} (\ell', Z')$ if $(\ell, g, a, Y, \ell') \in E$ and $Z' = ((Z \cap [\![g]\!])[Y])^{\nearrow}$. The simulation graph $SG(A)$ of A is defined as the transition system $(Z(Q), S_0, \to)$, where $Z(Q)$ is the set of zones of Q, $S_0 = ((\{\ell_0, \mathbf{0}\}^{\nearrow}) \cap [\![\mathsf{Inv}(\ell_0)]\!]$ and $\to$ defined as above.

Definition 2 (Timed Game Automaton [16]). *A* Timed Game Automaton (TGA) G *is a timed automaton with its set of actions* Act *partitioned into* controllable *(*Act_c*) and* uncontrollable *(*Act_u*) actions.*

2.2 Safety and Reachability Games

Given a TGA G and a set of states $K \subseteq L \times \mathbb{R}^X_{\geq 0}$ the *reachability control problem* consists in finding a *strategy* f s.t. G supervised by f enforces K. The *safety control problem* is the dual asking for the strategy to constantly avoid K. By "a reachability game (G, K)" (*resp.* safety) we refer to the reachability (*resp.* safety) control problem for G and K.

Let (G, K) be a reachability (resp. safety) game. A finite or infinite (ruling out runs with an infinite number of consecutive time transitions of duration 0) run $\rho = (\ell_0, v_0) \xrightarrow{e_0} (\ell_1, v_1) \xrightarrow{e_1} \cdots \xrightarrow{e_n} (\ell_{n+1}, v_{n+1}) \cdots$ in $\mathsf{Runs}(G)$ is *winning* if there is some $k \geq 0$ *s.t.* $(\ell_k, v_k) \in K$ (resp. for all $k \geq 0$ *s.t.* $(\ell_k, v_k) \in K$). The set of winning runs in G from (ℓ, v) is denoted $\mathsf{WinRuns}((\ell, v), G)$.

For reachability games we assume w.l.o.g. that the goal is a particular location Goal. For safety games the goal is a set a locations to avoid.

The formal definition of the control problems is based on the definitions of *strategies* and *outcomes*. A strategy [16] is a function that during the course of the game constantly gives information as to what the controller should do in order to win the game. In a given situation, the strategy could suggest the controller to either i) "do a particular controllable action" or ii) "do nothing at this point in time, just wait" which will be denoted by the special symbol λ.

Definition 3 (Strategy). *Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA. A* strategy *f over G is a partial function from $\mathsf{Runs}(G)$ to $\mathsf{Act}_c \cup \{\lambda\}$* s.t. *for every finite run ρ, if $f(\rho) \in \mathsf{Act}_c$ then $last(\rho) \xrightarrow{f(\rho)}_{S_G} (\ell', v')$ for some (ℓ', v').*

We denote $\mathsf{Strat}(G)$ the set of strategies over G. A strategy f is *state-based* whenever $\forall \rho, \rho' \in \mathsf{Runs}(G), last(\rho) = last(\rho')$ implies that $f(\rho) = f(\rho')$. State-based strategies are also called *memoryless* strategies in game theory [9,19].

The restricted behavior of a TGA G controlled with some strategy f is defined by the notion of *outcome* [9].

Definition 4 (Outcome). *Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA and f a strategy over G. The* outcome *$\mathsf{Outcome}(q, f)$ of f from q in S_G is the subset of $\mathsf{Runs}(q, G)$ defined inductively by:*

- *$q \in \mathsf{Outcome}(q, f)$,*
- *if $\rho \in \mathsf{Outcome}(q, f)$ then $\rho' = \rho \xrightarrow{e} q' \in \mathsf{Outcome}(q, f)$ if $\rho' \in \mathsf{Runs}(q, G)$ and one of the following three conditions hold:*
  1. *$e \in \mathsf{Act}_u$,*
  2. *$e \in \mathsf{Act}_c$ and $e = f(\rho)$,*
  3. *$e \in \mathbb{R}_{\geq 0}$ and $\forall 0 \leq e' < e, \exists q'' \in Q$ s.t. $last(\rho) \xrightarrow{e'} q'' \wedge f(\rho \xrightarrow{e'} q'') = \lambda$.*
- *for an infinite run ρ, $\rho \in \mathsf{Outcome}(q, f)$ if all the finite prefixes of ρ are in $\mathsf{Outcome}(q, f)$.*

We assume that uncontrollable actions can only spoil the game and the controller has to do some controllable action to win [6,16,11]. In other words, an uncontrollable action cannot be forced to happen in G. Thus, a run may end in a state where only uncontrollable actions can be taken. Moreover we focus on reachability games and assume $K = \{\mathsf{Goal}\} \times \mathbb{R}_{\geq 0}^X$. A *maximal run* ρ is either an infinite run (supposing no infinite sequence of delay transitions of duration 0) or a finite run ρ that satisfies either i) $last(\rho) \in K$ or ii) if $\rho \xrightarrow{a}$ then $a \in \mathsf{Act}_u$ (*i.e.* the only possible next discrete actions from $last(\rho)$, if any, are uncontrollable actions).

A strategy f is *winning* from q if all maximal runs in $\mathsf{Outcome}(q, f)$ are in $\mathsf{WinRuns}(q, G)$. A state q in a TGA G is *winning* if there exists a winning strategy f from q in G. We denote by $\mathcal{W}(G)$ the set of winning states in G and $\mathsf{WinStrat}(q, G)$ the set of winning strategies from q over G.

2.3 Backwards Algorithms for Solving Timed Games

Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA. For reachability games, the computation of the winning states is based on the definition of a *controllable predecessor* operator [9,16]. The controllable and uncontrollable discrete predecessors of X are defined by $\mathsf{cPred}(X) = \bigcup_{c\in\mathsf{Act}_c} \mathsf{Pred}_c(X)$ and $\mathsf{uPred}(X) = \bigcup_{u\in\mathsf{Act}_u} \mathsf{Pred}_u(X)$. A notion of *safe* timed predecessors of a set X *w.r.t.* a set Y is also needed. Intuitively a state q is in $\mathsf{Pred}_t(X, Y)$ if from q we can reach $q' \in X$ by time elapsing and along the path from q to q' we avoid Y. Formally this is defined by:

$$\mathsf{Pred}_t(X, Y) = \{q \in Q \mid \exists \delta \in \mathbb{R}_{\geq 0} \ s.t. \ q \xrightarrow{\delta} q', q' \in X \text{ and } \mathsf{Post}_{[0,\delta]}(q) \subseteq \overline{Y}\} \quad (1)$$

where $\mathsf{Post}_{[0,\delta]}(q) = \{q' \in Q \mid \exists t \in [0, \delta] \ s.t. \ q \xrightarrow{t} q'\}$ and $\overline{Y} = Q \setminus Y$. The *controllable predecessors* operator π is defined as follows[5]:

$$\pi(X) = \mathsf{Pred}_t\left(X \cup \mathsf{cPred}(X), \mathsf{uPred}(\overline{X})\right) \quad (2)$$

Let (G, K) be a reachability game, if S is a finite union of symbolic states, then $\pi(S)$ is again a finite union of symbolic states. Moreover the iterative process given by $W^0 = K$ and $W^{n+1} = \pi(W^n)$ will converge after finitely many steps for TGA [16] and the least fixed point obtained is W^*. It is also proved in [16] that $W^* = \mathcal{W}(G)$. Note also that W^* is the maximal set of winning states of G *i.e.* a state is winning iff it is in W^*. Thus there is a winning strategy in G iff $(\ell_0, \mathbf{0}) \in W^*$. Altogether this gives a symbolic algorithm for solving reachability games. Extracting strategies can be done using the winning set of states W^*. For safety games (G, K), it suffices to swap the roles of the players leading to a game $\overline{G}$ and solve a reachability game $(\overline{G}, \overline{K})$. If the winning set of states for $(\overline{G}, \overline{K})$ is W then the winning set of states of (G, K) is $\overline{W}$.

3 On-the-Fly Algorithm for Untimed Games

For finite-state systems, on-the-fly model-checking algorithms has been an active and successful research area since the end of the 80's, with the algorithm proposed by Liu & Smolka [15] being particularly elegant (and optimal). We present here our instantiation of this algorithm to untimed reachability games.

We consider untimed games as a restricted class of timed games with only finitely many states Q and with only discrete actions, *i.e.* the set of labels is Act. Hence (memoryless) strategies simplifies to a choice of controllable action given the current state, *i.e.* $f : Q \to \mathsf{Act}_c$. For (untimed) *reachability games* we assume a designated set Goal of goal-states and the purpose of the analysis is to decide the existence of a strategy f where all runs contains at least one state from Goal

[5] Note that π is defined here such that uncontrollable actions cannot be used to win.

```
Initialization:
    $Passed \leftarrow \{q_0\}$;
    $Waiting \leftarrow \{(q_0, \alpha, q') \mid \alpha \in Act\ q \xrightarrow{\alpha} q'\}$;
    $Win[q_0] \leftarrow (q_0 \in \mathsf{Goal}\ ?\ 1 : 0)$;
    $Depend[q_0] \leftarrow \emptyset$;

Main:
while $((Waiting \neq \emptyset) \wedge Win[q_0] \neq 1))$ do
    $e = (q, \alpha, q') \leftarrow pop(Waiting)$;
    if $q' \notin Passed$ then
        $Passed \leftarrow Passed \cup \{q'\}$;
        $Depend[q'] \leftarrow \{(q, \alpha, q')\}$;
        $Win[q'] \leftarrow (q' \in \mathsf{Goal}\ ?\ 1 : 0)$;
        $Waiting \leftarrow Waiting \cup \{(q', \alpha, q'') \mid q' \xrightarrow{\alpha} q''\}$;
        if $Win[q']$ then $Waiting \leftarrow Waiting \cup \{e\}$;
    else (* reevaluate *)
        $Win^* \leftarrow \bigwedge_{q \xrightarrow{u} u} Win[u] \wedge \bigvee_{q \xrightarrow{c} w} Win[w]$;
        if $Win^*$ then
            $Waiting \leftarrow Waiting \cup Depend[q]$; $Win[q] \leftarrow 1$;
        if $Win[q'] = 0$ then $Depend[q'] \leftarrow Depend[q'] \cup \{e\}$;
    endif
endwhile
```

Fig. 3. OTFUR: **O**n-**T**he-**F**ly Algorithm for **U**ntimed **R**eachability Games

(once again, safety games are solved by swapping the roles of the controller and the environment).

Now, our instantiation OTFUR of the local algorithm by Liu & Smolka to untimed reachability games is given in Fig. 3. This algorithm is based on a waiting-list, $Waiting \subseteq E$ of *edges* waiting to be explored together with a passed-list $Passed \subseteq Q$ containing the states that have been encountered so far. Information about the current winning status of a state is given by a function $Win : Passed \rightarrow \{0, 1\}$, where $Win[q]$ is initialized to 0 and later potentially upgraded to 1 when the winning status of successors to q change from 0 to 1. To activate the reevaluation of the winning status of states, each state q has an associated set of edges $Depend[q]$ *depending* on it: at any stage $Depend[q]$ contains all edges (q', α, q) that were encountered at a moment when $Win[q] = 0$ and where the winning status of the source state q' must be scheduled for reevaluation when $Win[q] = 1$ becomes true. We refer to [15] for the formal proof of correctness of this algorithm summarized by the following theorem:

Theorem 1 ([15]). *Upon termination of running the algorithm* OTFUR *on a given untimed game G the following holds:*

1. *If $q \in Passed$ and $Win[q] = 1$ then $q \in \mathcal{W}(G)$;*
2. *If $Waiting = \emptyset$ and $Win[q] = 0$ then $q \notin \mathcal{W}(G)$.*

In fact, the first property is an invariant of the **while**-statement holding after each iteration. Also, the algorithm is optimal in that it has linear time complexity in the size of the underlying untimed game: it is easy to see that each edge $e = (q, \alpha, q')$ will be added to $Waiting$ at most twice, the first time q is encountered (and added to *Passed*) and the second time when $Win[q']$ changes winning status from 0 to 1[6].

4 On-the-Fly Algorithm for Timed Games

Now let us turn our attention to the timed case and present our symbolic extension of the algorithm of Liu & Smolka providing a zone-based forward and on-the-fly algorithm for solving timed reachability games. The algorithm, SOTFTR, is given in Fig. 4 and may be viewed as an interleaved combination of *forward computation* of the *simulation graph* of the timed game automaton together with *back-propagation* of information *of winning states*. As in the untimed case the algorithm is based on a waiting-list, $Waiting$, of edges in the simulation-graph to be explored, and a passed-list, *Passed*, containing all the symbolic states of the simulation-graph encountered so far by the algorithm.

The crucial point of our symbolic extension is that the winning status $Win[q]$ of an individual state q is replaced by a set $Win[S] \subseteq S$ identifying the *subset* of the symbolic state S which is currently known to be winning. The set *Depend*$[S]$ indicates the set of edges (or predecessors of S) which must be reevaluated (*i.e.* added to $Waiting$) when new information about $Win[S]$ is obtained (*i.e.* when $Win[S] \subsetneq Win^*$). Whenever an edge $e = (S, \alpha, S')$ is considered with $S' \in Passed$, the edge e is added to the dependency set of S' in order that possible future information about additional winning states within S' may also be back-propagated to S. In Table 1, we illustrate the forward exploration and backwards propagation steps of the algorithm.

The correctness of the symbolic on-the-fly algorithm SOTFTR is given by the following lemma and theorem, the rigorous proofs of which can be found in the appendix.

Lemma 1. *The* **while**-*loop of algorithm* SOTFTR *has the following invariance properties when running on a timed game automaton G:*

1. *For any $S \in Passed$ if $S \xrightarrow{\alpha} S'$ then either $(S, \alpha, S') \in Waiting$ or $S' \in Passed$ and $(S, \alpha, S') \in Depend[S']$*
2. *If $q \in Win[S]$ for some $S \in Passed$ then $q \in \mathcal{W}(G)$*
3. *If $q \in S \setminus Win[S]$ for some $S \in Passed$ then either*
   - *$e \in Waiting$ for some $e = (S, \alpha, S')$ with $S' \in Passed$,* *or*
   - *$q \notin \mathsf{Pred}_t\big[Win[S] \cup \bigcup_{S \xrightarrow{c} T} \mathsf{Pred}_c(Win[T]), \bigcup_{S \xrightarrow{u} T} \mathsf{Pred}_u(T \setminus Win[T])\big]$.*

[6] To obtain an algorithm running in linear time in the size of G (*i.e.* $|Q| + |E|$) it is important that the reevaluation of the winning status of a state q does not directly involve (repeated and expensive) evaluation of the large boolean expression for Win^*. In a practice, this may be avoided by adding a boolean b_q and a counter c_q recording the existence of a winning, controllable successor of q, and the number of winning, uncontrollable successor of q.

```
Initialization:
    $Passed \leftarrow \{S_0\}$ where $S_0 = \{(\ell_0, \mathbf{0})\}^{\nearrow}$;
    $Waiting \leftarrow \{(S_0, \alpha, S') \mid S' = \mathsf{Post}_\alpha(S_0)^{\nearrow}\}$;
    $Win[S_0] \leftarrow S_0 \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
    $Depend[S_0] \leftarrow \emptyset$;

Main:
while $((Waiting \neq \emptyset) \wedge (s_0 \notin Win[S_0]))$ do
    $e = (S, \alpha, S') \leftarrow pop(Waiting)$;
    if $S' \notin Passed$ then
        $Passed \leftarrow Passed \cup \{S'\}$;
        $Depend[S'] \leftarrow \{(S, \alpha, S')\}$;
        $Win[S'] \leftarrow S' \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
        $Waiting \leftarrow Waiting \cup \{(S', \alpha, S'') \mid S'' = \mathsf{Post}_\alpha(S')^{\nearrow}\}$;
        if $Win[S'] \neq \emptyset$ then $Waiting \leftarrow Waiting \cup \{e\}$;
    else (* reevaluate *)[a]
        $Win^* \leftarrow \mathsf{Pred}_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} \mathsf{Pred}_c(Win[T]),$
                                  $\bigcup_{S \xrightarrow{u} T} \mathsf{Pred}_u(T \setminus Win[T])) \cap S$;
        if $(Win[S] \subsetneq Win^*)$ then
            $Waiting \leftarrow Waiting \cup Depend[S]$; $Win[S] \leftarrow Win^*$;
        $Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;
    endif
endwhile
```

[a] When $T \notin Passed, Win[T] = \emptyset$

Fig. 4. SOTFTR: **S**ymbolic **O**n-**T**he-**F**ly Algorithm for **T**imed **R**eachability Games

Theorem 2. *Upon termination of running the algorithm* SOTFTR *on a given timed game automaton G the following holds:*

1. *If $q \in Win[S]$ for some $S \in Passed$ then $q \in \mathcal{W}(G)$;*
2. *If $Waiting = \emptyset$, $q \in S \setminus Win[S]$ for some $S \in Passed$ then $q \notin \mathcal{W}(G)$.*

Termination of the algorithm SOTFTR is guaranteed by the finiteness of symbolic states[7] and the fact that each edge (S, α, T) will be present in the $Waiting$-list at most $1 + |T|$ times, where $|T|$ is the number of regions of T: (S, α, T) will be in $Waiting$ the first time that S is encountered and subsequently each time the value of $Win[T]$ increases. Now, any given region may be contained in several symbolic states of the simulation graph (due to overlap). Thus the SOTFTR algorithm is *not* linear in the region-graph and hence not theoretically optimal, as an algorithm with linear worst-case time-complexity could be obtained by applying the untimed algorithm directly to the region-graph. However, this is only a theoretical result and, as we shall see, the use of

[7] Strictly speaking, this requires that we either transforms the given TGA into an equivalent one in which all location-invariants insist on an upper bound on all clocks or, alternatively, that we apply standard extrapolation *w.r.t.* maximal constant occurring in the TGA (which is correct up to time-abstracted bisimulation).

Table 1. Running SOTFTG

Steps			Waiting	Passed	*Depend*	*Win*
#	S	S'				
0	-	-	$(S_0,u_1,S_1),(S_0,u_2,S_2),(\mathbf{S_0},\mathbf{c_1},\mathbf{S_3})$	S_0	-	$(S_0,\emptyset)$
1	S_0	S_3	$(S_0,u_1,S_1),(S_0,u_2,S_2)$ $+\ (\mathbf{S_3},\mathbf{c_2},\mathbf{S_4}),(S_3,u_3,S_2)$	S_3	$S_3\mapsto(S_0,c_1,S_3)$	$(S_3,\emptyset)$
2	S_3	S_4	$(S_0,u_1,S_1),(S_0,u_2,S_2),(S_3,u_3,S_2)$ $+\ (\mathbf{S_3},\mathbf{c_2},\mathbf{S_4})$	S_4	$S_4\mapsto(S_3,c_2,S_4)$	(S_4,S_4)
3	S_3	S_4	$(S_0,u_1,S_1),(S_0,u_2,S_2),(S_3,u_3,S_2)$ $+\ (\mathbf{S_0},\mathbf{c_1},\mathbf{S_3})$	-	-	$(S_3,x\geq 1)$
4	S_0	S_3	$(S_0,u_1,S_1),(S_0,u_2,S_2),(\mathbf{S_3},\mathbf{u_3},\mathbf{S_2})$	S_4	$S_3\mapsto(S_0,c_1,S_3)$	$(S_0,x=1)$
5	S_3	S_2	$(S_0,u_1,S_1),(S_0,u_2,S_2)$ $+\ (\mathbf{S_2},\mathbf{c_3},\mathbf{S_5})$	S_2	$S_2\mapsto(S_3,u_3,S_2)$	$(S_2,\emptyset)$
6	S_2	S_5	$(S_0,u_1,S_1),(S_0,u_2,S_2)$ $+\ (\mathbf{S_5},\mathbf{c_4},\mathbf{S_3})$	S_5	$S_5\mapsto(S_2,c_3,S_2)$	$(S_5,\emptyset)$
7	S_5	S_3	$(S_0,u_1,S_1),(S_0,u_2,S_2)$ $+\ (\mathbf{S_2},\mathbf{c_3},\mathbf{S_5})$	-	$S_3\mapsto\begin{matrix}(S_2,c_3,S_2)\\(S_5,c_4,S_3)\end{matrix}$	$(S_5,x\leq 1)$
8	S_2	S_5	$(S_0,u_1,S_1),(S_0,u_2,S_2)$ $+\ (\mathbf{S_3},\mathbf{u_3},\mathbf{S_2})$	-	$S_5\mapsto(S_2,c_3,S_2)$	$(S_2,x\leq 1)$
9	S_3	S_2	$(S_0,u_1,S_1),(\mathbf{S_0},\mathbf{u_2},\mathbf{S_2})$ $+\ (S_0,c_1,S_3),(S_5,c_4,S_3)$	-	-	(S_3,S_3)
10	S_0	S_2	$(S_0,u_1,S_1),(S_0,c_1,S_3),(\mathbf{S_5},\mathbf{c_4},\mathbf{S_3})$	-	$S_2\mapsto\begin{matrix}(S_3,u_3,S_2)\\(S_0,u_2,S_2)\end{matrix}$	$(S_0,x\leq 1)$
11	S_5	S_3	$(S_0,u_1,S_1),(\mathbf{S_0},\mathbf{c_1},\mathbf{S_3})$	-	-	-
12	S_0	S_3	$(\mathbf{S_0},\mathbf{u_1},\mathbf{S_1})$	-	-	-
13	S_0	S_1	$\emptyset$	S_1	$S_1\mapsto(S_0,u_1,S_1)$	$(S_1,\emptyset)$

At step n, $(\mathbf{S},\alpha,\mathbf{S'})$ is the transition popped at step $n+1$;
At step n, $+(S,\alpha,S')$ the transition added to *Waiting* at step n;
Symbolic States: $S_0=(\ell_1,x\geq 0)$,$S_1=(\ell_5,x>1)$, $S_2=(\ell_3,x\geq 0)$, $S_3=(\ell_2,x\geq 0)$, $S_4=(\mathsf{Goal},x\geq 2)$, $S_5=(\ell_4,x\geq 0)$

zones yields very encouraging performance results in practice, as is the case for reachability analysis of timed automata.

5 Implementation, Optimizations and Extensions

5.1 Implementation of the Pred_t Operator with Zones

In order to be efficient, the algorithm SOTFTR manipulates zones. However, while a forward step always gives a single zone as a result, the Pred_t operator does not. So, given a symbolic state S, $Win[S]$ is, in general, an union of zones (and so is $S\setminus Win[S]$). As a consequence, we now give two results, which allow us to handle unions of zones (Theorem 3) and to define the computation of Pred_t in terms of basic operations on zones (Theorem 4).

Theorem 3. *The following distribution law holds:*

$$\mathsf{Pred}_t(\bigcup_i G_i,\bigcup_j B_j)=\bigcup_i\bigcap_j \mathsf{Pred}_t(G_i,B_j) \tag{3}$$

Theorem 4. *If B is a convex set, then the* Pred_t *operator defined in equation (1) can be expressed as:*

$$\mathsf{Pred}_t(G,B)=(G^{\swarrow}\setminus B^{\swarrow})\cup((G\cap B^{\swarrow})\setminus B)^{\swarrow} \tag{4}$$

5.2 Optimizations

Zone Inclusion. When we explore forward the automaton, we check if any newly generated symbolic state S' belongs to the passed list: $S' \in \mathit{Passed}$. As an optimization we may instead use the classical inclusion check: $\exists S'' \in \mathit{Passed}$ s.t. $S' \subseteq S''$, in which case, S' is discarded and we update the dependency graph as well. Indeed, new information learned from the successors of S'' can be new information on S' but not necessarily. This introduces an overhead in the sense that we may back-propagate information for nothing.

On the other hand, back-propagating only the relevant information would be unnecessarily complex and would void most of the memory gain introduced by the use of inclusion. In practice, the reduction of the number of forward steps obtained by the inclusion check pays off for large systems and is a little overhead otherwise, as shown in our experiments.

Losing States. In the case of *reachability*, our games being determined, we can sometimes decide at an early stage that a state q is losing (*i.e.* $q \notin \mathcal{W}(S_G)$), either because it is given as a part of the model in the same way as goal states, or because it is deadlock state, which is not in the set of goal states.

The detection of such losing states has a two-fold benefit. First, we can stop the forward exploration on these states, since we know that we have lost (in the case of a user-defined non-deadlock losing state). Second, we can back-propagate these losing states in the same way as we do for winning states and stop the algorithm if we have the initial state $s_0 \in \mathit{Lose}[S_0]$, where $\mathit{Lose}[S]$ is the subset of the symbolic state S currently known to be losing. In some cases, this can bring a big benefit, illustrated by Fig. 1, if the guard $x < 1$ is changed to true in the edge from ℓ_1 to ℓ_5.

Pruning. In the basic algorithm early termination takes place when the initial state is known to be winning (*i.e.* $s_0 \in \mathit{Win}[S_0]$). However, we may extend this principle to other parts of the algorithm. In particular, we can add the condition that whenever an edge $e = (S, \alpha, S')$ is selected and it turns out that $\mathit{Win}[S] = S$ then we may safely skip the rest of the **while** loop as we know that no further knowledge on the winning states of S can be gained. In doing so, we prune unnecessary continued forward exploration and/or expensive reevaluation. When we back-propagate losing states as described previously, the condition naturally extends to $\mathit{Win}[S] \cup \mathit{Lose}[S] = S$.

5.3 Time Optimal Strategy Synthesis

Time-optimality for reachability games consists in computing the best (optimal) time the controller can guarantee to reach the Goal location: if t^* is the optimal-time, the controller has a strategy that guarantees to reach location Goal within t^* time units whatever the opponent is doing, and moreover, the controller has no strategy to guarantee this for any $t < t^*$.

First consider the following problem: decide whether the controller has a strategy to reach location Goal within B time units. To solve this problem, we

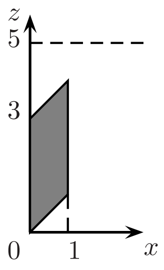


Fig. 5. Winning subset of the initial zone of the TGA of Fig. 1 with clock z added

just add a fresh clock z to the TGA G and the invariant $\mathsf{Inv}(\ell) \equiv z \leq B$ for all locations ℓ with z being unconstrained in the initial state. Then we compute the set of winning states of this game and check that $(\ell_0, \mathbf{0}, z = 0)$ is actually a winning state. If not, try with some $B' > B$. Otherwise we know that the controller can guarantee to reach Goal within B time units ... but in addition we have the optimal-time to reach Goal[8]. Indeed, when computing the winning set of states W^* on the TGA G augmented with the z clock (being initially unconstrained), we have the maximal set of winning states. This means that we obtain some $(\ell_0, Z_0) \in W^*$ and $(\ell_0, \mathbf{0}, z = 0) \in (\ell_0, Z_0)$. But $Z_0 \cap \{(\ell_0, \mathbf{0})\}$ gives us for free the optimal-time to reach Goal. Assume $I = Z_0 \cap \{(\ell_0, \mathbf{0})\}$, then $0 \in I$ and the upper bound of I is less than B. This means that starting in $(\ell_0, \mathbf{0})$ with $z \in I$ the controller can guarantee to reach Goal within B time units. And as W^* is the maximal set of winning states, starting with $z \notin I$ cannot guarantee this any more. Assume $I = [0, b]$. The optimal-time is then $t^* = B - b$. If it turns out that I is right open $[0, b[$, we even know more: that this optimal time t^* cannot be achieved by any strategy, but we can reach Goal in a time arbitrarily close to t^*. On the example of Fig. 1, if we choose $B = 5$ we obtain a closed interval $I = [0, 3]$ giving the optimal time $t^* = 2$ to reach Goal (Fig. 5). Moreover we know that there is a strategy that guarantees this optimal.

6 Experiments

Several versions of the described timed game reachability algorithm have been implemented: with or without inclusion checking between zones, with or without back-propagation of the losing states, and with or without pruning. To benchmark the implementations we used the Production Cell [14,17] case study (Fig. 6). Unprocessed plates arrive on a feeding belt, are taken by a robot to a press, are processed, and are taken away to a departure belt. The robot has two arms (A and B) to take and release the plates and its actions are controllable, except for the time needed to rotate. The arrival of the plates and the press are uncontrollable.

We run experiments on a dual-Xeon 2.8GHz equipped with 3GB of RAM running Linux 2.4.21. Table 2 shows the obtained results. The tests are done with varying number of plates from 2 to 7, and with controllable (win) and uncontrollable (lose) configurations. The models contain one clock for the controller and one clock for each plate. An extra clock is added in the case of timed optimal strategy.

[8] To get an optimum, the condition of the **while**-loop must be $Waiting \neq \emptyset$ alone in the algorithm, disabling early termination.

The inclusion checking of zones is shown to be an important optimization. Furthermore, activating pruning, which really exploits that the algorithm is *on-the-fly*, is useful in practice: the algorithm really terminates earlier. The results for time optimal reachability confirm that the algorithm is exploring the whole state-space and is comparable to exploring without pruning. We stress in the tables the best result obtained for every configuration: it turns out that propagating back the losing states has a significant overhead that pays off for large systems, *i.e.* it is clearly better from 6 plates.

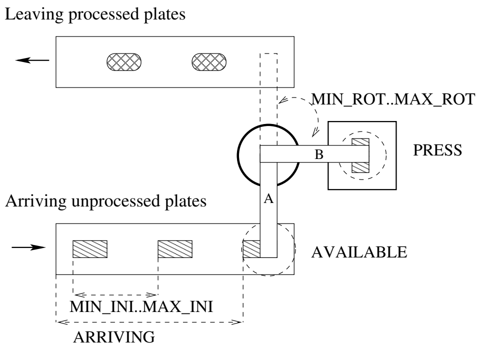


Fig. 6. The production cell

The state-space grows exponentially with respect to the number of plates but the algorithm keeps up linearly with it, which is shown on Fig. 7 that depicts $pre + post$[9]. These results show that the algorithm based on zones behaves well despite the fact that zones are (in theory) worse than Alur & Dill's regions.

Table 2. Results for the different implementations: basic algorithm, then with inclusion checking (inc), pruning (pruning), back propagation of losing states (backlose) and time optimal strategy generation (topt, only for "win", and pruning has little effect). For each number of plates, the tests are done with a controllable (win) and an uncontrollable (lose) configuration. Time (user process) is given in seconds (s) rounded to 0.1s and memory in megabytes (M). '-' denotes a failed run (not enough memory). Results in bold font are the best ones.

Plates		Basic		Basic +inc		Basic +inc +pruning		Basic +backlose +inc +pruning		Basic+backlose +inc +topt	
		time	mem	time	mem	time	mem	time	mem	time	mem
2	win	0.0s	1M	0.0s	1M	**0.0s**	**1M**	0.0s	1M	0.04s	1M
	lose	0.0s	1M	0.0s	1M	**0.0s**	**1M**	0.0s	1M	n/a	n/a
3	win	0.5s	19M	0.0s	1M	**0.0s**	**1M**	0.1s	1M	0.27s	4M
	lose	1.1s	45M	0.1s	1M	**0.0s**	**1M**	0.2s	3M	n/a	n/a
4	win	33.9s	1395M	0.2s	8M	**0.1s**	6M	0.4s	**5M**	1.88s	13M
	lose	-	-	0.5s	11M	**0.4s**	10M	0.9s	**9M**	n/a	n/a
5	win	-	-	3.0s	31M	**1.5s**	22M	2.0s	**16M**	13.35s	59M
	lose	-	-	11.1s	61M	**5.9s**	46M	7.0s	**41M**	n/a	n/a
6	win	-	-	89.1s	179M	38.9s	121M	**12.0s**	**63M**	220.3s	369M
	lose	-	-	699s	480M	317s	346M	**135.1s**	**273M**	n/a	n/a
7	win	-	-	3256s	1183M	1181s	786M	**124s**	**319M**	6188s	2457M
	lose	-	-	-	-	16791s	2981M	**4075s**	**2090M**	n/a	n/a

7 Conclusion and Future Work

In this paper we have introduced what we believe is the first completely on-the-fly algorithm for solving timed games. For its efficient implementation we have

[9] $pre + post$ represents the number of iterations of the algorithm and is therefore an abstraction in both time and space of the implementation.

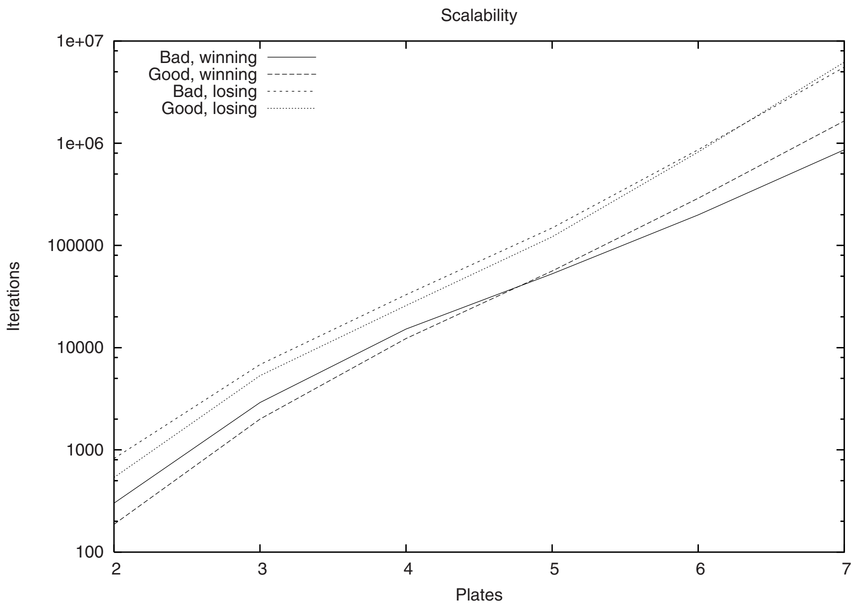


Fig. 7. Scalability of the algorithm. The scale is logarithmic.

used zones as the main datastructure, and we have applied decisive optimizations to take full advantage of the on-the-fly nature of our algorithm and its zone representation. Experiments have shown that an implementation based on zones is feasible and with encouraging performances *w.r.t.* the complexity of the problem. Finally, we have exhibited how to obtain the time optimal strategies with minor additions to our algorithm (essentially book-keeping).

We are working on an improved version of the implementation to distribute it and use the UPPAAL GUI augmented with (un)controllable transitions. We are investigating more aggressive abstractions for the underlying simulation graph computed by our algorithm and efficient guiding of the search, in particular for optimal strategies. Our algorithm is well suited for distributed implementation by its use of unordered waiting-list and there are plans to pursue this directions as has been done for UPPAAL [7]. We are also investigating how to extract strategies and represent them compactly with CDDs (Clock Decision Diagrams).

Acknowledgments. The authors want to thank Patricia Bouyer and Gerd Behrmann for inspiring discussions on the topic of timed games.

References

1. K. Altisen and S. Tripakis. Tools for controller synthesis of timed systems. In *Proc. 2nd Work. on Real-Time Tools (RT-TOOLS'02)*, 2002. Proc. published as Technical Report 2002-025, Uppsala University, Sweden.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

3. R. Alur, S. La Torre, and G. J. Pappas. Optimal Paths in Weighted Timed Automata. In *Proc. of 4th Work. Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *LNCS*, pages 49–62. Springer, 2001.
4. Henrik R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.
5. E. Asarin and O. Maler. As Soon as Possible: Time Optimal Control for Timed Automata. In *Proc. 2nd Work. Hybrid Systems: Computation & Control (HSCC'99)*, volume 1569 of *LNCS*, pages 19–30. Springer, 1999.
6. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller Synthesis for Timed Automata. In *Proc. IFAC Symp. on System Structure & Control*, pages 469–474. Elsevier Science, 1998.
7. Gerd Behrmann. Distributed reachability analysis in timed automata. *Journal of Software Tools for Technology Transfer (STTT)*, 7(1):19–30, 2005.
8. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: a Model-Checking Tool for Real-Time Systems. In *Proc. 10th Conf. on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.
9. L. De Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic Algorithms for Infinite-State Games. In *Proc. 12th Conf. on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
10. Gerard J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2003.
11. S. La Torre, S. Mukhopadhyay, and A. Murano. Optimal-Reachability and Control for Acyclic Weighted Timed Automata. In *Proc. 2nd IFIP Conf. on Theoretical Computer Science (TCS 2002)*, volume 223, pages 485–497. Kluwer, 2002.
12. K. G. Larsen. Efficient Local Correctness Checking. In *Proc. of Conf. of Computer Assisted Verification (CAV'92)*, volume 663 of *LNCS*, pages 30–43. Springer, 1992.
13. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal of Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.
14. C. Lewerentz and T. Lindner. Production Cell: A Comparative Study in Formal Specification and Verification. In *Methods, Languages & Tools for Construction of Correct Software*, volume 1009 of *LNCS*, pages 388–416. Springer, 1995.
15. X. Liu and S. Smolka. Simple Linear-Time Algorithm for Minimal Fixed Points. In *Proc. 26th Conf. on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
16. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. 12th Symp. on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 229–242. Springer, 1995.
17. H. Melcher and K. Winkelmann. Controller Synthesis for the "Production Cell" Case Study. In *Proc. of 2nd Work. on Formal Methods in Software Practice*, pages 24–36. ACM Press, 1998.
18. J. Rasmussen, K. G. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *Proc. 10th Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 220–235. Springer, 2004.
19. W. Thomas. On the Synthesis of Strategies in Infinite Games. In *Proc. 12th Symp. on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 1–13. Springer, 1995. Invited talk.
20. S. Tripakis and K. Altisen. On-the-Fly Controller Synthesis for Discrete and Timed Systems. In *Proc. of World Congress on Formal Methods (FM'99)*, volume 1708 of *LNCS*, pages 233–252. Springer, 1999.