Automatic Verification of Multi-queue Discrete Timed Automata

Pierluigi San Pietro[1]* and Zhe Dang[2]

[1] Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italia
pierluigi.sanpietro@polimi.it
[2] School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164, USA
zdang@eecs.wsu.edu

Abstract. We propose a new infinite-state model, called the Multi-queue Discrete Timed Automaton *MQDTA*, which extends Timed Automata with queues, but only has integer-valued clocks. Due to careful restrictions on queue usage, the binary reachability (the set of all pairs of configurations (α, β) of an *MQDTA* such that α can reach β through zero or more transitions) is effectively semilinear. We then prove the decidability of a class of Presburger formulae defined over the binary reachability, allowing the automatic verification of many interesting properties of a *MQDTA*. The *MQDTA* model can be used to specify and verify various systems with unbounded queues, such as a real-time scheduler.

Keywords: Timed Automata, infinite-state model-checking, real-time systems.

1 Introduction

Real-time systems are widely regarded as a natural application area of formal methods, since the presence of the time variable makes them more difficult to specify, design and test. The limited expressiveness of finite automata has recently sparkled much research into the automated verification of infinite state systems. Most research in the field has concentrated on finding good abstractions or approximations that map infinite state systems into finite ones (e.g., parametrized model checking [19] and generalized model checking [15]). A complementary approach to abstraction is the definition and study of infinite-state models for which "interesting" properties are still decidable. Most of the works have concentrated on very few models, such as Petri Nets (*PN*), Pushdown Automata (*PA*) and Timed Automata (*TA*), and have studied the decidability and complexity of model-checking various temporal and modal logics. A *TA* [4] is basically a finite-state automaton with a certain number of unbounded clocks that can be tested and reset. Since their introduction and the definition of appropriate model checking algorithms [17], *TA* have become a useful model to investigate the verification of real-time systems and have been extensively studied. The expressive power of *TA* has many limitations in modeling, since many real-time systems are simply not finite-state, even when time is ignored.

* Supported in part by MIUR grants FIRB RBAU01MCAC and COFIN 2001015271.

Other infinite-state models for which forms of automatic verification are possible are based on *PN* (e.g., [16]), on various versions of counter machines (e.g., [10]), on *PA* (e.g., [5]), or on process calculi (e.g., [21]), but, at least in their basic versions, they do not consider timing requirements and are thus not amenable for modeling real-time systems. Among the infinite-state models that consider time, there are many timed extensions of Petri Nets but their binary reachability is typically undecidable if the net is unbounded (i.e., it is not finite state). A recent notable example of model checking a timed version of Petri Nets is [2], where it is shown that coverability properties are decidable, using well-quasi orderings techniques. A more general result holds for an extension of *TA*, Timed Networks [1], for which safety properties have been shown to be decidable. However, Timed Networks consist of an arbitrary set of identical timed automata, which is a very special case, although potentially useful in modeling infinite-state timed systems.

Recently, Timed Pushdown Automata (*TPA*) [13, 12] have been proposed, extending pushdown processes with unbounded discrete clocks. Considering that both the region techniques [4] and the flattening techniques [11] for *TA* can not be used for *TPA*, a totally different technique is proposed to show that safety and binary reachability analysis are still decidable [13, 12].

Queues are a good model of many interesting systems, such as schedulers, for which automatic verification has rarely been attempted. Queues are usually regarded as hopeless for verification, since it is well known that a finite-state automaton equipped with one unbounded queue can simulate a Turing machine. However, there are restricted models with queues for which reachability is decidable (e.g., [9]). Here, we consider the Generalized Context-free Grammars (GCG) of [8], which use both queues and stacks with suitable constraints to generate only semilinear languages, and which are well suited to modeling of scheduling policies. However, automatic verification of GCG has never been investigated, and GCG do not consider time.

In this paper, we study how to couple a timed automaton with a multi-queue automaton (inspired by the GCG model) so that the resulting machine can be effectively used for modeling, while retaining the decidability of a class of Presburger formulae over the binary reachability set, with control-state variables, clock value variables and count variables. Hence, such machines are amenable for modeling and automatic verification of many infinite-state real-time systems, such as real-time process schedulers. The paper is structured as follows. Section 2 defines the *MQDTA*, introduces its untimed version (called Multi-queue-stack machine, *MQSM*) and proves the effective semilinearity of the model, by using a GCG. Section 3 proves the main result of the paper, i.e., the effective semilinearity of the binary reachability for *MQDTA*, by showing that clocks may be eliminated and an *MQDTA* may be translated into an equivalent *MQSM*. Section 4 proves the decidability of a class of Presburger formulae over the binary reachability, showing their applicability to an example.

2 Multi-queue Discrete Timed Automata

In this section we introduce the *MQDTA* model, which extends Discrete Timed Automata *DTA* by allowing a number of queues. The presentation is self-contained abd does not require previous knowledge of *DTA*. A *clock constraint* is a Boolean combination of *atomic clock constraints* in the following form: $x\#c, x - y\#c$ where c is an integer, x, y are integer-valued clocks and $\#$ denotes $\leq, \geq, <, >$, or $=$. Let $\mathcal{L}_X$ be the set of all clock constraints on clocks X. Let $\mathbf{Z}$ be the set of integers and $\mathbf{Z}^+$ be the set of nonnegative integers.

Definition 1 **(*MQDTA*).** *A Multi-queue Discrete Timed Automaton (MQDTA) with* $n \geq 0$ FIFO *queues is a tuple* $\langle S, X, \Gamma, s_f, \mathbf{R}, E, Q_1, \cdots, Q_n \rangle$ *where:* Γ *is the queue alphabet;* $Q_1, \cdots, Q_n$ *are queues;* S *is a finite set of (control) states;* X *is a finite set of clocks with values in* $\mathbf{Z}^+$*;* $s_f \in S$ *is the final state;* $\mathbf{R} \subseteq \Gamma \times S$ *is the restart set;* E *is a finite set of edges, such that each edge* $e \in E$ *is in the form of* $\langle s, \lambda, (\eta_1, \cdots, \eta_n), l, s' \rangle$ *where* $s, s' \in S$ *with* $s \neq s_f$ *(the final state* s_f *does not have a successor);* $\lambda \subseteq X$ *is the set of clock resets;* $l \in \mathcal{L}_X$ *is the enabling condition.*

The queue operation is characterized by a tuple $(\eta_1, \cdots, \eta_n)$ *with* $\eta_1, \cdots, \eta_n \in \Gamma^*$*, to denote that each* η_i *is put at the end of the queue* Q_i*,* $1 \leq i \leq n$*.*

Let $\mathcal{A}$ be an *MQDTA* with n queues. Intuitively, the queues are totally ordered from Q_1 to Q_n and for a pair $(\gamma, s) \in \mathbf{R}$, s will be the next start state of $\mathcal{A}$ if the head of the first nonempty queue is γ. Notice that, for $n = 0$, the *MQDTA* reduces to a *DTA*.

The semantics is defined as follows. A *configuration* α of $\mathcal{A}$ is a tuple $\langle s, \pi_1, \cdots, \pi_n, c_1, \cdots, c_k \rangle$ where $s \in S, c_1, \cdots, c_k \in \mathbf{Z}^+$ are the state and the clock values respectively. $\pi_1, \cdots, \pi_n \in \Gamma^*$ are the contents of each queue, with the leftmost character being the head and rightmost character being the tail. We use α_{Q_i} to denote each π_i in α, with $\alpha_{\mathbf{q}}$, $\alpha_{x_1}, \cdots, \alpha_{x_k}$ to denote $s, c_1, \ldots, c_k$ respectively.

Let $\alpha \xrightarrow{\langle s, \lambda, (\eta_1, \cdots, \eta_n), l, s' \rangle} \alpha'$ denote a one-step transition along an edge $\langle s, \lambda, (\eta_1, \cdots, \eta_n), l, s' \rangle$ in $\mathcal{A}$ satisfying the following conditions:

– The state s is set to a new location s', i.e., $\alpha_{\mathbf{q}} = s, \alpha'_{\mathbf{q}} = s'$.
– Each clock changes according to λ. If there are no clock resets on the edge, i.e., $\lambda = \emptyset$, then clocks progress by one time unit, i.e., for each $x \in X$, $\alpha'_x = \alpha_x + 1$. If $\lambda \neq \emptyset$, then for each $x \in \lambda$, $\alpha'_x = 0$ while for each $x \notin \lambda$, $\alpha'_x = \alpha_x$.
– The enabling condition is satisfied, i.e., $l(\alpha)$ is true.
– The content of each queue is updated: $\alpha'_{Q_i} = \alpha_{Q_i}\eta_i$ for each $1 \leq i \leq n$.

Besides the above defined one-step transition, an *MQDTA* $\mathcal{A}$ can fire a restart transition when it is in the final state s_f. Let $\alpha \xrightarrow{restart} \alpha'$ denote a *restart transition* in $\mathcal{A}$ satisfying the following four conditions:
1) $\alpha_{\mathbf{q}} = s_f$, i.e., this restart transition only fires at the final state.
2) Some queue in α is not empty. Let $\gamma \in \Gamma$ be the head of the first (in the order from Q_1 to Q_n) nonempty queue. The next state should be indicated in the restart set $\mathbf{R}$. That is, $(\gamma, \alpha'_{\mathbf{q}}) \in \mathbf{R}$.
3) Let γ be the head of the j-th queue where $1 \leq j \leq n$ and α_{Q_j} is not empty, and for all $1 \leq i < j$, α_{Q_i} is empty. Assume $\alpha_{Q_j} = \gamma\pi$ for some $\pi \in \Gamma^*$. Then, $\alpha'_{Q_j} = \pi$, and for all $1 \leq i \leq n$ with $i \neq j$, $\alpha'_{Q_i} = \alpha_{Q_i}$. That is, the head γ must be removed from the queue, while the other queues are not modified.
4) Clocks are reset, i.e., $\alpha'_x = 0$ for all $x \in X$.

From now on, $\mathcal{A}$ is a *MQDTA* specified as above. We simply write $\alpha \rightarrow_{\mathcal{A}} \alpha'$ if α can reach α' by either a one-step transition or a restart transition. The *binary reachability* $\rightsquigarrow^{\mathcal{A}}$ is the reflexive and transitive closure of $\rightarrow_{\mathcal{A}}$ A configuration $\alpha = \langle s, \pi_1, \cdots,$

$\pi_n, c_1, \cdots, c_k\rangle$ can be encoded as a string $[\alpha]$ by concatenating the symbol representation of s, the strings $\pi_1, \cdots, \pi_n$, and the (unary) string representation of $c_1, \cdots, c_k$ with a delimiter "\$". The binary reachability $\rightsquigarrow^{\mathcal{A}}$ can be considered as the language: $\{[\alpha]\$[\beta] : \alpha \rightsquigarrow^{\mathcal{A}} \beta\}$.

An example. Consider a LAN printer, which may accept two types of jobs: Large (L) and Small (S). When a job is being printed, no other job can interrupt it. However, if a job takes too long to be completed, then the printer preempts it and puts it into a special, lower priority queue, called the batch queue. The timeout for the L jobs is 200 seconds, while for the S jobs is only 100. The jobs in the batch queue (called batch jobs, B) can be printed without time limits, but they are overridden (i.e., put at the end of the batch queue) whenever L or S job arrives. The arrival of new jobs is not completely random: if the printer is busy printing, then the interval between the arrival of new jobs is at least 50 seconds. The specifi cation of the example is formalized with an *MQDTA* with two clocks (called $timeout$ and $last$ respectively) and two queues. The set of states is: $\{start, printL, printS, printB\}$. The alphabet of the queues is: $\{L, S, B\}$. The graph of the transition function is shown in Fig. 1. Multiple transitions from one state to another state are denoted by multiple labels instead of by multiple arcs. The labels used on the transitions have the following syntax: [clock condition] / [queue update] [clock assignment]. The notation for clock conditions and assignments is obvious. A queue update such as $(L)1(B)2$ means that L and B are written on queue 1 and queue 2, respectively. The automaton starts the execution in the $start$ state. When either an S or an L job is put into queue 1, the automaton enters s_f: it reads the queue content and executes a $restart$ transition (denoted by the dashed arrows). A $restart$ transition goes from s_f to the next state depending on the queue content: if the front of the queue is S it enters state $printS$, if it is L it enters state $printL$, if it is B it enters state $printB$. When a $restart$ transition is executed, all the clocks are reset (i.e., $timeout := 0$ and $last := 0$ are executed).

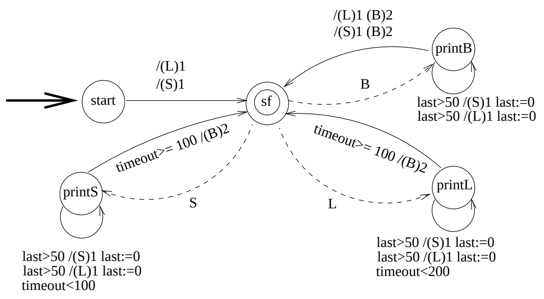

Fig. 1. The state transition graph of the $DTMQ$ $\mathcal{A}$ of the example.

The proof of the main result of this paper, i.e., $\rightsquigarrow^{\mathcal{A}}$ is a semilinear language, is based on the following untimed version of *MQDTA*, which allows both queues and stacks. A *Multi-queue-Stack Machine (MQSM)* M with n (*FIFO*) queues, m (LIFO) stacks and a one-way input tape is a tuple: $\langle S, \Sigma, \Gamma, \Theta, s_0, s_f, \delta, Q_1, \cdots, Q_n, C_1, \cdots, C_m\rangle$, where S is a fi nite set of states with the *initial state* $s_0 \in S$ and the *final state* $s_f \in S$, Σ

is the input alphabet, Γ and Θ are two disjoint alphabets for the queues $Q_1, \cdots, Q_n$ and the stacks $C_1, \cdots, C_m$ respectively. The queues and the stacks are arranged so that $Q_1, \cdots, Q_n$ are followed by $C_1, \cdots, C_m$. δ is a finite set of *transitions*. We distinguish three kinds of transitions:

A *push-transition* has the form $\langle s, \sigma, (\eta_1, \cdots, \eta_n, \xi_1, \cdots, \xi_m), s' \rangle$. That is, from state $s \in S$, M moves its input head to the right and reads an input symbol $\sigma \in \Sigma$ (if $\sigma = \epsilon$, however, M the input head does not move, i.e., M executes an ϵ-move), puts $\eta_1, \cdots, \eta_n \in \Gamma^*$ at the end of queues $Q_1, \cdots, Q_n$, and pushes $\xi_1, \cdots, \xi_m \in \Theta^*$ into the stacks $C_1, \cdots, C_m$. Thus, a push-transition is an element of $S \times (\Sigma \cup \{\epsilon\}) \times (\Gamma^*)^n \times (\Theta^*)^m \times S$. A *pop-stack-transition* has the form $\langle s_f, \sigma, \theta, s' \rangle$. That is, from the final state $s_f \in S$, on the input symbol $\sigma \in \Sigma \cup \{\epsilon\}$, M pops the top of the first nonempty stack and transits to state $s' \in S$. Thus, a pop-stack-transition is an element of $\{s_f\} \times (\Sigma \cup \{\epsilon\}) \times \Theta \times S$. A *pop-queue-transition* has the form of $\langle s_f, \sigma, \gamma, s' \rangle$. That is, from the final state s_f, on the input symbol $\sigma \in \Sigma \cup \{\epsilon\}$, M pops the top of the first nonempty queue and transits to state $s' \in S$. Thus, a pop-queue-transition is an element of $\{s_f\} \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times S$. Therefore, δ is a finite subset of $(S \times (\Sigma \cup \{\epsilon\}) \times (\Gamma^*)^n \times (\Theta^*)^m \times S) \cup (\{s_f\} \times (\Sigma \cup \{\epsilon\}) \times \Theta \times S) \cup (\{s_f\} \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times S)$.

A *configuration* of M is a tuple $\langle s, w; \gamma_1, \cdots, \gamma_n; c_1, \cdots, c_m \rangle$, where $s \in S$ is the state, $w \in \Sigma^*$ is the input word, $\gamma_1, \cdots, \gamma_n \in \Gamma^*$ are the contents of the queues (with the leftmost character being the head and the rightmost character being the tail), $c_1, \cdots, c_m \in \Theta^*$ are the contents of the stacks (with the leftmost character being the top and rightmost character being the bottom). The one-step transition $\Rightarrow_M$ of M is a binary relation over configurations. That is,

$$\langle s, w; \gamma_1, \cdots, \gamma_n; c_1, \cdots, c_m \rangle \Rightarrow_M \langle s', w'; \gamma'_1, \cdots, \gamma'_n; c'_1, \cdots, c'_m \rangle$$

iff one of the conditions is satisfied:

– The transition is a push-transition $\langle s, \sigma, (\eta_1, \cdots, \eta_n, \xi_1, \cdots, \xi_m), s' \rangle \in \delta$. Then: $w = \sigma w'$, for each $1 \le i \le n$, $\gamma'_i = \gamma_i \eta_i$, and for each $1 \le j \le m$, $c'_j = \xi_j c_j$.
– The transition is a pop-stack-transition $\langle s_f, \sigma, \theta, s' \rangle \in \delta$. Then: $s = s_f$, $w = \sigma w'$, for each $1 \le i \le n$, $\gamma'_i = \gamma_i = \epsilon$. There exists $1 \le j \le m$ such that $c_j = \theta c'_j$ and for all $1 \le k \le j - 1$, $c'_k = c_k = \epsilon$, and for all $j + 1 \le k \le m$, $c'_k = c_k$.
– The transition is a pop-queue-transition $\langle s_f, \sigma, \gamma, s' \rangle \in \delta$. Then: $s = s_f$, $w = \sigma w'$, there is a $1 \le i \le n$ such that $\gamma_i = \gamma \gamma'_i$, for all $1 \le k \le i - 1$, $\gamma_k = \gamma'_k = \epsilon$, for all $i + 1 \le k \le n$, $\gamma_k = \gamma'_k$, and for all $1 \le k \le n$, $c'_k = c_k$.

The *transition relation* $\Rightarrow^*_M$ is the transitive closure of the binary relation $\Rightarrow_M$ over configurations. A string $w \in \Sigma^*$ is *accepted* by M if $\langle s_0, w; \gamma_0, \epsilon, \cdots, \epsilon; \epsilon, \cdots, \epsilon \rangle \Rightarrow^*_M \langle s_f, \epsilon; \epsilon, \cdots, \epsilon; \epsilon, \cdots, \epsilon \rangle$.

Theorem 1. *Languages accepted by Multi-queue-stack machines are semilinear.*

Proof. The proof of this result is based on the fact that the language accepted by *MQSM* may be generated by a Generalized Context-free Grammar. In fact, an *MQSM* is actually a GCG in disguise, and it is just a variant of the accepting device of GCG, the multi-queue-pushdown automaton [7]. GCG only generate suitable permutations of context-free languages, and hence their languages are semilinear and the semilinear sets are effectively constructible. □

3 Main Results

To prove the main result of the paper, we need a few more definitions. Let $\Sigma = \{a_1, \ldots, a_n\}$ be an alphabet, for some $n \geq 1$. A *Parikh transform* P translates each $w \in \Sigma^*$ into $a_1^{\#_{a_1}(w)} \cdots a_k^{\#_{a_k}(w)}$. Let $\$ \notin \Sigma$ be a symbol. For every $n \geq 1$, a language L is *segmented* if every word of L has the form $w_1\$w_2 \cdots \$w_n$, where each $w_i \in \Sigma^*$. P is abused on every word $w = w_1\$ \cdots \$w_n$ of a segmented language L, with $P(\mathbf{w}) = P(w_1)\$ \cdots \$P(w_n)$, and $P(L) = \{P(\mathbf{w}) : \mathbf{w} \in L\}$. A segmented language L is *locally commutative* if $\mathbf{w} \in L$ iff $P(\mathbf{w}) \in P(L)$.

Lemma 1. *For all languages L_1 and L_2, with L_2 segmented and locally commutative, the following statements hold: 1. $P(L_1)$ is a semilinear language iff L_1 is. 2. If L_1 and L_2 are semilinear languages then so is $L_1 \cap L_2$.*

Proof. *1.* Let p be the Parikh mapping (see [18] for this traditional definition). Simply notice that $p(P(L_1)) = p(L_1)$. *2.* Suppose that each word L_2 has $n \geq 1$ occurrences of \$, and let C be the language of all the words in $(\Sigma \cup \{\$\})^*$ with exactly n occurrences of \$. Let L_3 be the segmented language $L_1 \cap C$, which is semilinear because L_1 is semilinear and C is obviously semilinear and commutative. Since $L_2 \subseteq C$, then $L_1 \cap L_2 = L_1 \cap (C \cap L_2) = L_3 \cap L_2$. Hence, $P(L_1 \cap L_2) = P(L_3 \cap L_2) = P(L_3) \cap P(L_2)$, since L_2 is locally commutative and L_3 is segmented. Then, from the proof of part (1) above, $p(L_1 \cap L_2) = p(P(L_1 \cap L_2)) = p(P(L_3) \cap P(L_2))$. Since elements in $P(L_3)$ and $P(L_2)$ are made of tuples, and from part (1), $P(L_3)$ and $P(L_2)$ are semilinear languages, also $P(L_3) \cap P(L_2)$ is a semilinear language [14]. The result follows from part (1). □

Lemma 2. *Let $L \subseteq \{[\alpha]\$[\beta] : \alpha, \beta$ are configurations of $\mathcal{A}\}$ be a semilinear language. Then, given a clock constraint $l \in \mathcal{L}_X$, $L' := \{[\alpha]\$[\beta] \in L : l(\alpha_{x_1}, \cdots, \alpha_{x_k})\}$ is also a semilinear language.*

The proof of Lemma 2 is immediate from Lemma 1.

Let $\mathcal{A}$ = be an *MQDTA* with clocks $x_1, \cdots, x_k$ and queues $Q_1, \cdots, Q_n$. We now show a technique to eliminate the *tests* (which are the enabling conditions, i.e., Boolean combinations of $x_i\#c$, $x_i - x_j\#c$ with c an integer constant) in an *MQDTA* $\mathcal{A}$. Let m be one plus the maximal absolute value of all the integer constants that appear in the tests in $\mathcal{A}$. Denote the finite set $[m] =_{def} \{-m, \cdots, 0, \cdots, m\}$. *Entries* a_{ij} and b_i for $1 \leq i, j \leq k$ are finite state variables with values in $[m]$. Intuitively, a_{ij} and b_i are used to record $x_i - x_j$ and x_i respectively. However, during the computation of $\mathcal{A}$, when $x_i - x_j$ (or x_i) goes beyond m or below $-m$, a_{ij} (or b_i) stays the same as m or $-m$. On executing an edge e, the new entry values a'_{ij} and b'_i can be expressed by only using the old values a_{ij} and b_i through *entry updating instructions*. For instance, suppose the set of clock resets is $\lambda = \{x_3\}$ for e. The entry updating instructions on this edge are: for all $1 \leq i, j \leq k$ with $i, j \neq 3$, $a'_{33} := 0; a'_{3j} = -b_j; a'_{i3} := b_i; a'_{ij} := a_{ij}; b'_i := b_i; b'_3 := 0$. The detailed construction of the entry updating instructions on any edge e is omitted, since it is a variant of the one presented in [13]. The addition of appropriate entry updating instructions on each edge guarantees that: after $\mathcal{A}$ executes any transition, (1). $x_i - x_j\#c$ iff $a_{ij}\#c$, (2). $x_i\#c$ iff $b_i\#c$, for all $1 \leq i, j \leq k$ and for each integer $-m < c < m$. The proof of this statement (omitted here) is similar to one in [13], though here we deal with different machine models and different clock behaviors. Thus, by adding entry updating instructions, each $x_i - x_j \# c$ (or $x_i \# c$) can be replaced by $a_{ij} \# c$ (or $b_i \# c$). The resulting automaton is denoted by $\mathcal{A}'$.

The *replaced tests* and the entry updating instructions in $\mathcal{A}'$ can be further eliminated by expanding the states S. The result automaton is called $\mathcal{A}^2$ with states $S_2 \subseteq S \times [m]^{(k^2+k)}$. In short, each expanded state in S_2 indicates the *original* state $s \in S$ and values (totally $k^2 + k$ many) of entries a_{ij} and b_i. Each edge e (connecting a pair of states s, s' in S) in $\mathcal{A}'$ is thus split into a fi nite number of edges in $\mathcal{A}^2$. Each split edge in $\mathcal{A}^2$ connects two expanded states $\bar{s}, \bar{s}'$ in S_2 with the their original states being s and s' respectively, and the values of the entries in $\bar{s}$ satisfying the test on e (thus the replaced tests are eliminated in $\mathcal{A}^2$), and the values of the entries in $\bar{s}$ and $\bar{s}'$ being consistent with the entry updating instructions on e (thus the entry updating instructions are eliminated in $\mathcal{A}^2$). Recall that, for an *MQDTA*, there is only one fi nal state s_f. So, in $\mathcal{A}^2$, all the expanded states with their original state being s_f are merged into one state s_f – doing this will not change the behavior of $\mathcal{A}^2$ since the fi nal state is used to initiate a restart transition and its enabling condition cannot depend on clock values. The restart set $\mathbf{R}_2$ of $\mathcal{A}^2$ is modifi ed as the set of all pairs $(\gamma, \bar{s})$ such that $(\gamma, s) \in \mathbf{R}$ in $\mathcal{A}'$ and $\bar{s}$ has the original state s and all the entry values in $\bar{s}$ are 0 (since after a restart transition from s_f to s, all the clocks $x_1, \cdots, x_k$ become zero.). Now, $\mathcal{A}^2$ is an *MQDTA* with states S_2, restart set $\mathbf{R}_2$, clocks $x_1, \cdots, x_k$, and queues $Q_1, \cdots, Q_n$. Each enabling condition in $\mathcal{A}^2$ is simply *true*.

Based upon the above simulations, we can establish the following theorem, noticing that $\mathcal{A}'$ simulates $\mathcal{A}$ as we indicated before.

Theorem 2. *For all configurations α and β of $\mathcal{A}$, $\alpha \leadsto^{\mathcal{A}} \beta$ iff there are configurations α^2 and β^2 of $\mathcal{A}^2$ such that $\alpha^2 \leadsto^{\mathcal{A}^2} \beta^2$, and the following conditions hold:*

- *The original states of the extended states $\alpha^2_{\mathbf{q}}$ and $\beta^2_{\mathbf{q}}$ are $\alpha_{\mathbf{q}}$ and $\beta_{\mathbf{q}}$ respectively. The entry values a_{ij} and b_i in the extended state $\alpha^2_{\mathbf{q}}$ are the initial entry values constructed from α as: if $|\alpha_{x_i} - \alpha_{x_j}| \le m$, $a_{ij} := \alpha_{x_i} - \alpha_{x_j}$; if $\alpha_{x_i} - \alpha_{x_j} > m$, $a_{ij} := m$; if $\alpha_{x_i} - \alpha_{x_j} < -m$, $a_{ij} := -m$; if $\alpha_{x_i} \le m$, $b_i := \alpha_{x_i}$; if $\alpha_{x_i} > m$, $b_i := m$; for each $1 \le i, j \le k$.*
- *Clock values and queue contents are the same in α and α^2, and in β and β^2.*

Theorem 3. *If $\leadsto^{\mathcal{A}^2}$ is semilinear, then so is $\leadsto^{\mathcal{A}}$.*

Proof. From Theorem 2, the entry values in the extended states $\alpha^2_{\mathbf{q}}$ and $\beta^2_{\mathbf{q}}$ can be dropped by applying a homomorphism. However, $\leadsto^{\mathcal{A}}$ is the homomorphic image not only of $\leadsto^{\mathcal{A}^2}$, but of a proper subset of $\leadsto^{\mathcal{A}^2}$. In fact, as stated in Theorem 2, the entry values in the extended state $\alpha^2_{\mathbf{q}}$ are the initial entry values constructed from α. This condition can be expressed as a clock constraint, since the entry values a_{ij} and b_i are bounded. The result is immediate by applying Lemma 2 on $\mathcal{A}^2$, and applying the homomorphism to L' as in the lemma. □

Theorem 4. *The language $\leadsto^{\mathcal{A}^2}$ is semilinear.*

Proof. The language $\leadsto^{\mathcal{A}^2}$ is $\{[\alpha]\$[\beta] : \alpha \leadsto^{\mathcal{A}^2} \beta\}$. An *MQSM* M to simulate $\mathcal{A}^2$ has an input alphabet including all the following symbols:

- symbols $\dot{s}$ and $\ddot{s}$ for each s in the state set of $\mathcal{A}^2$. $\dot{s}$ is used to encode the state $\alpha_{\mathbf{q}}$ of α, and $\ddot{s}$ is used to encode the state $\beta_{\mathbf{q}}$ of β.
- symbols $\dot{u}_i$ and $\ddot{u}_i$, $1 \leq i \leq k$. $\dot{u}_i$ is used to encode the unary string representation of the clock value α_{x_i}, and $\ddot{u}_i$ is for β_{x_i}.
- symbols $\dot{\gamma}$ and $\ddot{\gamma}$ for each $\gamma \in \mathit{\Gamma}$. Letters $\dot{\gamma}$ are used to encode queue words $\alpha_{Q_1}, \cdots, \alpha_{Q_n}$ of α. Letters $\ddot{\gamma}$ are for those of β.
- $3k + 2n + 3$ delimiters $\$, \dot{\&}, \ddot{\&}, \ddot{\&}_i, \dot{\#}_i, \dot{?}_j, \ddot{\#}_i, \ddot{?}_j$, for $1 \leq i \leq k$ and $1 \leq j \leq n$.
- *padding symbols* $\dot{@}$ and $\ddot{\%}_i$ for $1 \leq i \leq k$.

The format of the input to M is:

$$\alpha_{\mathbf{q}}\dot{\&}\dot{\alpha}_{Q_1}\dot{?}_1 \cdots \dot{\alpha}_{Q_n}\dot{?}_n \dot{u}_1^{\alpha_{x_1}}\dot{\#}_1 \cdots \dot{u}_k^{\alpha_{x_k}}\dot{\#}_k \dot{@}^t \$ \ddot{\beta}_{\mathbf{q}}\ddot{\&}\ddot{\beta}_{Q_1}\ddot{?}_1 \cdots \ddot{\beta}_{Q_n}\ddot{?}_n \ddot{u}_1^{\beta_{x_1}}\ddot{\&}_1\ddot{\%}_1^{t_1}\ddot{\#}_1 \cdots \ddot{u}_k^{\beta_{x_k}}\ddot{\&}_k\ddot{\%}_k^{t_k}\ddot{\#}_k$$

The part before \$ is the encoding for α, and the part after \$ is for the encoding for β. The first part has four segments, from left to right:

- $\dot{\alpha}_{\mathbf{q}}$ is a symbol encoding the state $\alpha_{\mathbf{q}}$, followed by a delimiter $\dot{\&}$.
- $\dot{\alpha}_{Q_1}\dot{?}_1 \cdots \dot{\alpha}_{Q_n}\dot{?}_n$ is the concatenation of the queue words α_{Q_i}, using the delimiters $\dot{?}_1, \cdots, \dot{?}_n$. Note that, instead of using α_{Q_i} for a queue word, we use $\dot{\alpha}_{Q_i}$ by replacing each $\gamma \in \mathit{\Gamma}$ with $\dot{\gamma}$.
- $\dot{u}_1^{\alpha_{x_1}}\dot{\#}_1 \cdots \dot{u}_k^{\alpha_{x_k}}\dot{\#}_k$ is the unary string representations $\dot{u}_i^{\alpha_{x_i}}$ of the clock value α_{x_i} using the symbol $\dot{u}_i$, concatenated by delimiters $\dot{\#}_1, \cdots, \dot{\#}_k$.
- a *padding word* $\dot{@}^t$ is a unary string over character $\dot{@}$. The number t is used to indicate the number of transitions in $\mathcal{A}^2$ that lead from α to β.

The second part has three segments from left to right, the first two being defined similarly, while the third one $\ddot{u}_1^{\beta_{x_1}}\ddot{\&}_1\ddot{\%}_1^{t_1}\ddot{\#}_1 \cdots \ddot{u}_k^{\beta_{x_k}}\ddot{\&}_k\ddot{\%}_k^{t_k}\ddot{\#}_k$ is the unary string representation $\ddot{u}_i^{\beta_{x_i}}$ of the clock value β_{x_i} using the symbol $\ddot{u}_i$, concatenated by delimiters $\ddot{\#}_1, \cdots, \ddot{\#}_k$. But we do not simply use $\ddot{u}_i^{\beta_{x_i}}$: instead, there is a padding $\ddot{\%}_i^{t_i}$ (a unary word of length t_i over the character $\ddot{\%}_i$) after each $\ddot{u}_i^{\beta_{x_i}}$, separated by a delimiter $\ddot{\&}_i$. These *clock padding words* will be made clear later.

Besides queues $Q_1, \cdots, Q_n$, M has stacks $C_1, \cdots, C_k$. Each stack C_i is used to store the clock value of x_i of $\mathcal{A}^2$. At start, M first pushes a new symbol Z_i twice onto each stack C_i – these symbols are used as indicate the bottom of each stack. M then reads the input tape up to the padding word $\dot{@}^t$. During the process, M stores the queue contents and clock values into $Q_1, \cdots, Q_n$ and $C_1, \cdots, C_k$ respectively. Then, M starts to simulate $\mathcal{A}^2$ from the state $\alpha_{\mathbf{q}}$ read from the input tape. Each move of $\mathcal{A}^2$ causes M to read a symbol @ and to simulate the queue operations using its own queues. The clock changes in $\mathcal{A}^2$ are simulated by using the stacks of M. Suppose that currently $\mathcal{A}^2$ executes an edge with a set λ of clock resets. If $\lambda = \emptyset$, then, after firing the transition, all clocks progress by one time unit. M simulates this by pushing the symbol $\ddot{u}_i$ onto the stack C_i for each $1 \leq i \leq k$. Otherwise, if $\lambda \neq \emptyset$, then, after the firing of the transition, the clocks in λ are reset and the others are left unchanged. M simulates this by pushing a special symbol Z_i onto the stack C_i for each $x_i \in \lambda$, and by pushing nothing (ϵ) on the other stacks. During the simulation of $\mathcal{A}^2$, M never pops the stacks, and in fact it need not, since all the enabling conditions in $\mathcal{A}^2$ are simply *true*. After having read all the padding word $@^t$, when M reads the delimiter \$, it must make sure that the current state of $\mathcal{A}^2$ corresponds to the symbol $\ddot{\beta}_{\mathbf{q}}$ on the input tape. M also pushes a new symbol Y_i onto each queue Q_i, in order to use them later to decide whether a queue is empty. M then moves to the final state s_f, which is also the final state of $\mathcal{A}^2$. There, M starts checking that the rest of the input tape is consistent with its current queue and stack contents. Such check requires M to pop repeatedly from its queues and stacks; these operations require, from the definition of an *MQSM*, that pop-queue-transitions and pop-stack-transitions occur in a final state and that the next state after a pop operation only depends on the current input character and the queue or stack symbol just read. But we use different sets of alphabets in the encoding of the rest of the input tape. Therefore, a pop-queue-transition executed now cannot be confused with a normal pop-queue-transition in M's simulating $\mathcal{A}^2$ when reading the padding word $\dot{@}^t$.

M proceeds by emptying each i-th queue, from Q_1 to Q_n, while checking the correspondence between the current top symbol of a queue and the symbol on the input tape. M can also check when the queue Q_i becomes empty by checking that the current input character is the delimiter $\ddot{?}_i$ and that the current top of the queue is Y_i (the symbol M pushed before). After all the queues are successfully compared and emptied, M starts to compare the clock values $\ddot{u}_i^{\beta_{x\,i}}$ on the input tape with the stack C_i, from C_1 to C_k. For each C_i, M reads the input $\ddot{u}_i^{\beta_{x\,i}}$ and pops a symbol from C_i. Once the bottom symbol Z_i becomes the current top symbol, the current input character must be the delimiter $\ddot{\&}_i$. After this, M empties C_i by reading through the clock padding word $\ddot{\%}_i^{t_i}$, but it makes sure that the delimiter $\ddot{\#}_i$, right after the clock padding word, is correspondent to the last symbol Z_i on the stack (remember that initially we pushed two Z_i's onto the stack. Thus, t_i is a guess of how many symbols there are between the first Z_i and the last Z_i in the stack. What if such a guess is wrong? In that case, since we use different $\ddot{u}_i$ for each i to represent both the stack word and the clock values on the input tape, M always knows, assuming the guess is wrong, whether a stack symbol $\ddot{u}_i$ hits an unexpected symbol like $\ddot{u}_j$ – either the guess of t_i is too small or it is too large. In this case, and in all the other cases where comparisons fail, M moves into a deadlock state – a special state where no further transition is possible.

M accepts the input iff all comparisons are successful and the input head is at the end of the tape. Notice that M has no ϵ-moves. Denote with $L(M)$ the language accepted by M. Thus, $L(M)$ is a semilinear language from Theorem 1. Notice that M does not check whether the input is in a correct format. Let L' be the regular language composed of all the strings in the correct format-it is a segmented language with $3k+2n+3$ segments. It is easy to check that L' is also a semilinear, locally commutative language. Thus, $L'' = L(M) \cap L'$, i.e., the set of all input strings accepted by M and in the correct format, is also a semilinear language from Lemma 1. L'' is different from the language $\rightsquigarrow^{\mathcal{A}^2}$, but not too much. From the previous construction, $\alpha \rightsquigarrow^{\mathcal{A}^2} \beta$ iff there are t, $t_1, \cdots, t_k$, such that the input word given as in the beginning of this proof can be accepted by M. Thus, define a homomorphism h such that, $h(\dot{@}) = h(\ddot{\%}_i) = \epsilon$, $h(\$) = h(\dot{\#}_i) = h(\dot{\&}) = h(\ddot{\&}) = h(\ddot{\&}_i) = h(\dot{?}_j) = h(\ddot{\#}_i) = h(\ddot{?}_j) = \$$, $h(\dot{s}) = h(\ddot{s}) = s$, $h(\dot{\gamma}) = h(\ddot{\gamma}) = \gamma, h(\dot{u}_i) = h(\ddot{u}_i) = 1$, for all $1 \leq i \leq k$ and $1 \leq j \leq n$, for all s being a state of $\mathcal{A}^2$, for all $\gamma \in \mathit{\Gamma}$. Obviously, $h(L'') = \rightsquigarrow^{\mathcal{A}^2}$. Thus,

$\rightsquigarrow^{\mathcal{A}^2}$ is a semilinear language (since the homomorphic image of a semilinear language is still semilinear). □

The following main theorem can be shown by combining Theorems 3 and 4.

Theorem 5. $\rightsquigarrow^{\mathcal{A}}$ *is a semilinear language for any* MQDTA $\mathcal{A}$.

An *MQDTA* $\mathcal{A}$ has no input tape, i.e., there is no event label on edges. However, if each edge is labeled, we can extend the states of $\mathcal{A}$ by combining a state with a label. In this case, a configuration contains only the current event label instead of the whole input word consumed. This may make applications more convenient to be dealt with, though all results still hold.

4 Verification Results

In this section, we formulate properties that can be verified for an *MQDTA*. Given an *MQDTA* $\mathcal{A}$, let $\alpha, \beta \cdots$ denote variables ranging over configurations, and let $\alpha_{\mathbf{q}}$ (state variables), α_{x_i} (clock value variables) and α_{Q_j} (queue content variables) denote, respectively, the state, the clock x_i's value and the content of the queue Q_j of α, $1 \le i \le k, 1 \le j \le n$. We use a count variable $\#_\gamma(\alpha_{Q_j})$ to denote the number of occurrences of a character $\gamma \in \Gamma$ in the content of the queue Q_j in α, $1 \le j \le n$. An *MQDTA*-term t is defined as follows: $t ::= n \mid \alpha_{x_i} \mid \#_\gamma(\alpha_{Q_j}) \mid t - t \mid t + t$, where n is an integer, $\gamma \in \Gamma$, $1 \le i \le k, 1 \le j \le n$. An *MQDTA*-formula f is defined as follows: $f ::= t > 0 \mid t \bmod n = 0 \mid \neg f \mid f \vee f \mid a_{\mathbf{q}} = q$, where $n \neq 0$ is an integer and q is a state of $\mathcal{A}$. Thus, f is a quantifier-free Presburger formula over control state variables, clock value variables and count variables. For $m \ge 1$, let F be a formula in the following format: $\bigvee_{1 \le i \le m}(f_i \wedge \alpha^i \rightsquigarrow^{\mathcal{A}} \beta^i)$, where each f_i is a *MQDTA*-formula and all α^i and β^i are configuration variables. Let $\exists F$ be a closed formula such that each free variable in F is existentially quantified. Then, the property $\exists F$ can be verified.

Theorem 6. *The truth value of* $\exists F$ *with respect to an* MQDTA $\mathcal{A}$ *is decidable for any MQDTA-formula* F.

Proof. Let $L(E)$ be the language of the string encodings of the tuples of all the configurations that satisfy a *MQDTA*-formula E. Thus, $L(F) = \bigcup_i \left(L(\alpha^i \rightsquigarrow^{\mathcal{A}} \beta^i) \cap L(f_i)\right)$. We will show that $L(F)$ is a semilinear language. Since all the proofs are constructive, the semilinear set of $L(F)$ can be effectively constructed from F and $\mathcal{A}$. Thus, testing whether $\exists F = false$, which is equivalent to testing the emptiness of $L(F)$, is decidable [14]. Since semilinearity is closed under union, without loss of generality we show that $L_1 \cap L_2$ is a semilinear language, where $L_1 = L(\alpha \rightsquigarrow^{\mathcal{A}} \beta)$ and $L_2 = L(f)$. From Theorem 5, L_1 is a semilinear language. Notice that L_2 is locally commutative – the reason is that the contents of the queues are used only by count variables. $P(L_2)$, the result of applying the Parikh transform P, is a semilinear language. Thus, from Lemma 1, part (1), L_2 is a semilinear language. Hence, $L_1 \cap L_2$ is a semilinear language from Lemma 1, part (2). □

For instance, the following property: "for all configurations α and β with $\alpha \rightsquigarrow^{\mathcal{A}} \beta$, clock x_2 in β is the sum of clocks x_1 and x_2 in α, and symbol γ_1 appears in the first queue Q_1 in β is twice as many as symbol γ_2 does in the second queue Q_2 in α." can be expressed as, $\forall\alpha\forall\beta(\alpha \rightsquigarrow^{\mathcal{A}} \beta \rightarrow (\beta_{x_2} = \alpha_{x_1} + \alpha_{x_2} \wedge \#_{\gamma_1}(\beta_{Q_1}) = 2\#_{\gamma_2}(\alpha_{Q_2})))$. The negation of this property is equivalent to $\exists F$ for some *MQDTA*-formula F. Thus, it can be verifi ed.

Verification of an example. Consider the LAN printer example of Section 2. A property verifi able with our model is that the fi rst queue is actually bounded: it can never contain more than 4 elements. This can be formalized as follows: for every α, β, such that $\alpha \rightsquigarrow^{\mathcal{A}} \beta$: $\alpha_{start} \wedge \#_S(\alpha_{Q1}) = \#_L(\alpha_{Q1}) = \#_B(\alpha_{Q2}) = 0 \wedge \#_S(\beta_{Q1}) + \#_L(\beta_{Q2}) \leq 4$. Notice that the binary queue of $\mathcal{A}$ need not be bounded, but the boundedness is a decidable property of $\mathcal{A}$. If this is the case, the implementation of the system might rely on a small buffer of size 4 to implement the queue. More sophisticated properties can also be verifi ed, and the system itself could be made more complex.

5 Conclusions

We introduced a new version of Timed Automata augmented with queues (*MQDTA*), and we proved that its binary reachability is effectively semilinear, allowing the automatic verifi cation of a class of Presburger formulae over control state variables, clock value variables and queue content.

An *MQDTA* is more powerful than the other timed (fi nite-state or pushdown) models, and it can be used for modeling various systems where *FIFO* policies are used. Such models are based on discrete–rather than dense–time. This choice is perfectly adequate for *synchronous real-time systems*, where there is always an underlying discrete model of time, but it is also suitable for modeling various asynchronous systems where discrete time is a good approximation of a dense one. Since this is the fi rst paper introducing and investigating the model, we did not develop explicitly a verifi cation algorithm. Using an automata-theoretic approach, we reduced the problem of checking reachability properties to checking certain Presburger formulae over integer values. Hence, the the complexity of the verifi cation has a very high upper bound (nondeterministic double exponential). This is not as hopeless as it may seem. For instance, the Omega-library of [20] could be used to implement a verifi cation algorithm, since the library is usually reasonably effi cient for formulae without alternating quantifi ers (as in our case). Also, a very high upper bound is typical of the automata-theoretic approach, but often the upper bound may be reduced by using a (more complex) process algebra approach.

The *MQDTA* has some limitations in expressivity: for instance, it cannot check how long a symbol has been stored in a queue before being consumed. Moreover, when a queue is read, all the clocks are reset. However, the model could be powerful enough to describe and verify useful, real-life infi nite-state systems (such as the simple job scheduler with timeouts and a priority queue of Section 2) that, at the best of our knowledge, cannot be modeled and automatically verifi ed by any other formalism. The model only considers queues, but in general stacks could be used instead or together with queues, since the GCG model (on which *MQDTA* are based) allows both kinds of rewriting policies. This can be useful, since for instance a stack can model recursive procedure calls, and the queues may model a process scheduler.

Our results imply that it is decidable to verify whether the paths of the reachability satisfy constraints expressed in a fragment of Presburger arithmetic. This can be easily achieved by recording, in one additional queue of the automaton, the history of the moves. For instance, it is possible to verify that the total time a symbol has been waiting in a queue does not exceed a given threshold, even though, as remarked above, this control cannot be done by the *MQDTA* itself.

References

1. P. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theoretical Computer Science*, 290(1):241–264, 2002.
2. P. Abdulla and A. Nyln. Timed petri nets and bqos. In *ICATPN'2001, 22nd Int. Conf. on application and theory of Petri nets*, 2001.
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
5. M. Benedikt P. Godefroid and T. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP 2001*, of *LNCS* 2076, pp. 652–666. Springer, 2001.
6. L. Breveglieri, A. Cherubini, C. Citrini, and S. Crespi Reghizzi. Multiple pushdown languages and grammars. *Int. Journal of Found. of Computer Science*, 7:253–291, 1996.
7. L. Breveglieri, A. Cherubini, and S. Crespi Reghizzi. Real-time scheduling by queue automata. In *FTRTFT'92*, vol/ 571 of *LNCS*, pages 131–148. Springer, 1992.
8. L. Breveglieri, A. Cherubini, and S. Crespi Reghizzi. Modelling operating systems schedulers with multi-stack-queue grammars. In *Fundamentals of Computation Theory*, volume 1684 of *LNCS*, pages 161–172. Springer, 1999.
9. G. Cece and A. Finkel. Programs with quasi-stable channels are effectively recognizable. In *CAV'97*, volume 1254 of *LNCS*, pages 304–315. Springer, 1997.
10. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV'98*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.
11. H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *CONCUR'99*, volume 1664 of *LNCS*, pages 242–257. Springer, 1999.
12. Zhe Dang. Binary reachability analysis of pushdown timed automata with dense clocks. In *CAV'01*, volume 2102 of *LNCS*, pages 506–517. Springer, 2001.
13. Zhe Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. In *CAV'00*, *LNCS* 1855, pages 69–84. Springer, 2000.
14. S. Ginsburg and E. Spanier. Semigroups, presburger formulas, and languages. *Pacifi c J. of Mathematics*, 16:285–296, 1966.
15. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *CAV'02*, volume 2404 of *LNCS*, pages 137–150. Springer, 2002.
16. B. Grahlmann. The state of pep. In *AMAST'98*, *LNCS* 1548, pages 522–526. Springer, 1998.
17. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.
18. R. Parikh. On context-free languages. *Journal of the ACM*, 13:570–581, 1966.
19. A. Pnueli and E. Shahar. Livenss and acceleraiton in parameterized verification. In *CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
20. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
21. B. Steffen and O.Burkart. Model checking the full modal mu-calculus for infinite sequential processes. In *ICALP'97*, volume 1256 of *LNCS*, pages 419–429.