Learning of Event-Recording Automata

Olga Grinchtein, Bengt Jonsson, and Martin Leucker*

Department of Computer Systems, Uppsala University, Sweden
{olgag, bengt, leucker}@it.uu.se

Abstract. We extend Angluin's algorithm for on-line learning of regular languages to the setting of *timed systems*. We consider systems that can be described by a class of deterministic *event-recording automata*. We present two algorithms that learn a description by asking a sequence of membership queries (does the system accept a given timed word?) and equivalence queries (is a hypothesized description equivalent to the correct one?). In the constructed description, states are identified by sequences of symbols; timing constraints on transitions are learned by adapting algorithms for learning hypercubes. The number of membership queries is polynomially in the minimal zone graph and in the biggest constant of the automaton to learn for the first algorithm. The second algorithm learns a (usually) smaller representation of the underlying system.

1 Introduction

Research during the last decades have developed powerful techniques for using *models of reactive systems* in specification, automated verification (e.g., [9]), test case generation (e.g., [12, 24]), implementation (e.g., [16]), and validation of reactive systems in telecommunication, embedded control, and related application areas. Typically, such models are assumed to be developed *a priori* during the specification and design phases of system development. In practice, however, often no formal specification is available, or becomes outdated as the system evolves over time. One must then construct a model that describes the behavior of an existing system or implementation. In software verification, techniques are being developed for generating abstract models of software modules by static analysis of source code (e.g., [10, 19]). However, peripheral hardware components, library modules, or third-party software systems do not allow static analysis. In practice, such systems must be analyzed by observing their external behavior. In fact, techniques for constructing models by analysis of externally observable behavior (black-box techniques) can be used in many situations.

- To create models of hardware components, library modules, that are part of a larger system which, e.g., is to be formally verified or analyzed.
- For regression testing, a model of an earlier version of an implemented system can be used to create a good test suite and test oracle for testing subsequent versions. This has been demonstrated, e.g., by Hungar et al. [15, 20]).

* This author is supported by the European Research Training Network "Games".

– Black-box techniques, such as adaptive model checking [14], have been developed to check correctness properties, even when source code or formal models are not available.
– Tools that analyze the source code statically depend heavily on the implementation language used. Black-box techniques are easier to adapt to modules written in different languages.

The construction of models from observations of system behavior can be seen as a learning problem. For finite-state reactive systems, it means to construct a (deterministic) finite automaton from the answers to a finite set of *membership queries*, each of which asks whether a certain word is accepted by the automaton or not. There are several techniques (e.g., [4, 13, 21, 23, 5]) which use essentially the same basic principles; they differ in how membership queries may be chosen and in exactly how an automaton is constructed from the answers. The techniques guarantee that a correct automaton will be constructed if "enough" information is obtained. In order to check this, Angluin and others also allow *equivalence queries* that ask whether a hypothesized automaton accepts the correct language; such a query is answered either by *yes* or by a counterexample on which the hypothesis and the correct language disagree. Techniques for learning finite automata have been successfully used for regression testing [15] and model checking [14] of finite-state systems for which no model or source code is available.

In this paper, we extend the learning algorithm of Angluin and others to the setting of timed systems. One longer-term goal is to develop techniques for creating abstract timed models of hardware components, device drivers, etc. for analysis of timed reactive systems; there are many other analogous applications. To the best of our knowledge, this is the first work on learning of timed systems; it is not an easy challenge, and we will therefore in this first work make some idealizing assumptions. We assume that a learning algorithm observes a system by checking whether certain actions can be performed at certain moments in time, and that the learner is able to control and record precisely the timing of the occurrence of each action. We consider systems that can be described by a timed automaton [2], i.e., a finite automaton equipped with clocks that constrain the possible absolute times of occurrences of actions. Since timed automata can not in general be determinized [2], we restrict consideration to a class of *event-recording automata* [3]. These are timed automata that, for every action a, use a clock that records the time of the last occurrence of a. Event-recording automata can be determinized, and are sufficiently expressive to model many interesting timed systems; for instance, they are as powerful as timed transition systems [17, 3], another popular model for timed systems.

In this work, we further restrict event-recording automata to be event-deterministic in the sense that each state has at most one outgoing transition per action (i.e., the automaton obtained by removing the clock constraints is deterministic). Under this restriction, timing constraints for the occurrence of an action depend only on the past sequence of actions, and not on their relative timing; learning such an automaton becomes significantly more tractable, and allows us to adapt the learning algorithm of Angluin to the timed setting.

We present two algorithms, LSGDERA and LDERA, for learning deterministic event-recording automata. LSGDERA learns a so-called *sharply guarded* deterministic event-recording automaton. We show that every deterministic event-recording automaton can be transformed into a unique sharply guarded one with at most double exponentially more locations. We then address the problem of learning a smaller, not necessarily sharply guarded version of the system. The algorithm LDERA achieves this goal by *unifying* the queried information when it is “similar” which results in merging states in the automaton construction.

We show that the number of membership queries of LSGDERA is polynomial in the size of the biggest constant appearing in guards and in the number n of locations of the sharply guarded deterministic event-recording automaton. Furthermore, we show that every deterministic event-recording automaton can be transformed into a sharply guarded one with at most double exponentially more locations. The number of equivalence queries is at most n. LDERA exceeds these bounds in the worst case, however, in practice it can be expected that it behaves better than LSGDERA.

We are not aware of any other work on learning of timed systems or timed languages. However, several papers are concerned with finding a definition of timed languages which is suitable as a basis for learning. There are several works that define determinizable classes of timed automata (e.g., [3, 25]) and right-congruences of timed languages (e.g., [22, 18, 26]), motivated by testing and verification.

The paper is structured as follows. After preliminaries in the next section, we define deterministic event-recording automata (DERA) in Section 3. In Section 4, we present our techniques for learning DERAs and their timing constraints. Section 5 gives a short example and shows the differences of both algorithms.

2 Preliminaries

We write $\mathbb{R}^{\geq 0}$ for the set of nonnegative real numbers, and $\mathbb{N}$ for the set of natural numbers. Let Σ be a finite alphabet of size $|\Sigma|$. A *timed word* over Σ is a finite sequence $w_t = (a_1, t_1)(a_2, t_2) \dots (a_n, t_n)$ of symbols $a_i \in \Sigma$ that are paired with nonnegative real numbers t_i such that the sequence $t_1 t_2 \dots t_n$ of time-stamps is nondecreasing. We use λ to denote the empty word. A *timed language* over Σ is a set of timed words over Σ.

An event-recording automaton contains for every symbol $a \in \Sigma$ a clock x_a, called the *event-recording clock* of a. Intuitively, x_a records the time elapsed since the last occurrence of the symbol a. We write C_Σ for the set $\{x_a | a \in \Sigma\}$ of event-recording clocks.

A *clock valuation* γ is a mapping from C_Σ to $\mathbb{R}^{\geq 0}$. A *clock constraint* is a conjunction of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C_\Sigma$, $\sim \in \{\leq, \geq\}$, and $n \in \mathbb{N}$. We use $\gamma \models \varphi$ to denote that the clock valuation γ satisfies the clock constraint φ. A clock constraint is *K-bounded* if it contains no constant larger than K. A clock constraint φ *identifies* a $|\Sigma|$-dimensional *polyhedron* $[\![\varphi]\!] \subseteq (\mathbb{R}^{\geq 0})^{|\Sigma|}$ viz. the vectors of real numbers satisfying the constraint. A *clock guard* is a clock constraint whose conjuncts are only of the form $x \sim n$ (for $x \in C_\Sigma$, $\sim \in \{\leq, \geq\}$), i.e., comparison between clocks is not permitted. The set of clock guards is denoted by G. A clock guard g *identifies* a $|\Sigma|$-dimensional *hypercube* $[\![g]\!] \subseteq (\mathbb{R}^{\geq 0})^{|\Sigma|}$. Thus, for every guard g that is satisfiable, we can talk of its *smallest corner*, denoted by $sc(g)$, using the notions from the cube identified by g. If furthermore K is the biggest constant appearing in g, we call a valuation γ a *biggest corner* of g, if γ is maximal in the dimensions where $[\![g]\!]$ is bounded and exceeds K in the others. The set of all biggest corners for a guard g is denoted by $bc(g)$. Sometimes, when convenient, we identify all values greater than K and denote them by ∞. Furthermore, we use *true* (*false*) to denote constraints that are always (never, respectively) satisfiable. Sometimes, the context requires $x = \mathit{true}$ to mean $x \geq 0 \wedge x \leq \infty$ and $x = \mathit{false}$ to mean $x \leq 0 \wedge x \geq \infty$.

Clock constraints can efficiently and uniquely be represented using difference bound matrices (DBMs, [11]). Furthermore, DBMs allow efficient operations on clock constraints like intersection, checking equality etc.

A *clocked word* w_c is a sequence $w_c = (a_1, \gamma_1)(a_2, \gamma_2) \ldots (a_n, \gamma_n)$ of symbols $a_i \in \Sigma$ that are paired with event-clock valuations. Each timed word $w_t = (a_1, t_1)(a_2, t_2) \ldots (a_n, t_n)$ can be naturally transformed into a clocked word $CW(w_t) = (a_1, \gamma_1)(a_2, \gamma_2) \ldots (a_n, \gamma_n)$ where for each i with $1 \leq i \leq n$,

- $\gamma_i(x_a) = t_i$ if $a_j \neq a$ for $1 \leq j < i$,
- $\gamma_i(x_a) = t_i - t_j$ if there is a j with $1 \leq j < i$ and $a_j = a$, such that $a_k \neq a$ for $j < k < i$.

A *guarded word* w_g is a sequence $w_g = (a_1, g_1)(a_2, g_2) \ldots (a_n, g_n)$ of symbols $a_i \in \Sigma$ that are paired with clock guards. Note that we identify an empty conjunction with *true*. For a clocked word $w_c = (a_1, \gamma_1)(a_2, \gamma_2) \ldots (a_n, \gamma_n)$ we use $w_c \models w_g$ to denote that $\gamma_i \models g_i$ for $1 \leq i \leq n$. For a timed word w_t we use $w_t \models w_g$ to denote that $CW(w_t) \models w_g$.

A guarded word $w_g = (a_1, g_1)(a_2, g_2) \ldots (a_n, g_n)$ is called a *guard refinement* of $a_1 a_2 \ldots a_n$, and $a_1 a_2 \ldots a_n$ is called the word *underlying* w_g. The word w underlying a timed word w_t is defined in a similar manner.

A *deterministic finite automaton* (DFA) $\mathcal{A} = \langle \Gamma, L, l_0, \delta \rangle$ over the alphabet Γ consists of states L, initial state l_0, and a partial transition function $\delta : L \times \Gamma \to L$. A *run* of $\mathcal{A}$ over the word $w = a_1 a_2 \ldots a_n$ is a finite sequence

$$l_0 \xrightarrow{a_1} l_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} l_n$$

of states $l_i \in L$ such that l_0 is the initial state and $\delta(l_{i-1}, a_i)$ is defined for $1 \leq i \leq n$, with $\delta(l_{i-1}, a_i) = l_i$. In this case, we write $\delta(l_0, w) = l_n$, thereby extending the definition of δ in the natural way. The language $\mathcal{L}(\mathcal{A})$ comprises all words $a_1 a_2 \ldots a_n$ over which a run exists.[1]

[1] Usually, DFAs are equipped with accepting states. We are only interested in prefix-closed languages. For these languages, DFAs with partial transition function and every state assumed to be accepting suffice.

3 Deterministic Event-recording automata

Definition 1. *A* deterministic event-recording automaton *(DERA)* $D = \langle \Sigma, L, l_0, \delta, \eta \rangle$ *consists of a finite* input alphabet Σ, *a finite set* L *of* locations, *an* initial location $l_0 \in L$, *a* transition function $\delta : L \times \Sigma \to L$, *which is a partial function that for each location and input symbol potentially prescribes a target location, a* guard function $\eta : L \times \Sigma \to G$, *which is a partial function that for each location and input symbol prescribes a clock guard, whenever* δ *is defined for this pair.*

In order to define the language accepted by a DERA, we first understand it as a DFA.

Given a DERA $D = \langle \Sigma, L, l_0, \delta, \eta \rangle$, we define $\mathit{dfa}(D)$ to be the DFA $\mathcal{A}_D = \langle \Gamma, L, l_0, \delta' \rangle$ over the alphabet $\Gamma = \Sigma \times G$ where $\delta' : L \times \Gamma \to L$ is defined by $\delta'(l, (a, g)) = \delta(l, a)$ if and only if $\delta(l, a)$ is defined and $\eta(l, a) = g$, otherwise $\delta'(l, (a, g))$ is undefined. Note that D and $\mathit{dfa}(D)$ have the same number of locations/states. Further, note that this mapping from DERAs over Σ to DFAs over $\Sigma \times G$ is injective, meaning that for each DFA $\mathcal{A}$ over $\Sigma \times G$, there is a unique (up to isomorphism) DERA over Σ, denoted $\mathit{dera}(\mathcal{A})$, such that $\mathit{dfa}(\mathit{dera}(\mathcal{A}))$ is isomorphic to $\mathcal{A}$.

The language $\mathcal{L}(D)$ accepted by a DERA D is defined to be the set of timed words w_t such that $w_t \models w_g$ for some guarded word $w_g \in \mathcal{L}(\mathit{dfa}(D))$. We call two DERAs D_1 and D_2 equivalent iff $\mathcal{L}(D_1) = \mathcal{L}(D_2)$, and denote this by $D_1 \equiv_t D_2$, or just $D_1 \equiv D_2$. A DERA is K-*bounded* if all its guards are K-bounded.

From the above definitions, we see that the language of a DERA D can be characterized by a prefix-closed set of guarded words $(a_1, g_1)(a_2, g_2) \ldots (a_n, g_n)$ in $\mathcal{L}(\mathit{dfa}(D))$ such that each $a_1 a_2 \ldots a_n$ occurs in at most one such guarded word. Thus, we can loosely say that D imposes on each untimed word $a_1 a_2 \ldots a_n$ the timing constraints represented by the conjunction of the guards $g_1 g_2 \ldots g_n$.

Example 1. The event-recording automaton shown in Figure 1 uses three event-recording clocks, x_a, x_b, and x_c. Location 0 is the start location of the automaton. Clock constraint $x_b \geq 3$ that is associated with the edge from location 1 to 4 ensures that the time difference between b and the subsequent a is greater or equal to 3. □

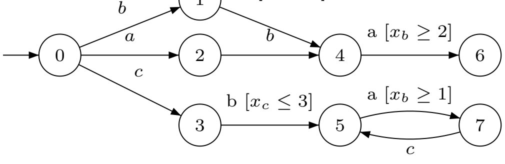


Fig. 1. An event-recording automaton

A central idea in Angluin's construction of finite automata is to let each state be identified by the words that reach it from the initial state (such words are called *access strings* in [5]). States are equivalent if, according to the queries submitted so far, the same continuations of their access strings are accepted. This idea is naturally based on the nice properties of Nerode's right congruence (given a language L, two words $u, v \in \Sigma^*$ are equivalent if for all $w \in \Sigma^*$ we have $uw \in L$ iff $vw \in L$) which implies that there is a unique minimal DFA accepting L. In other words, for DFAs, every state can be characterized by the set of words accepted by the DFA when considering this state as an initial state, and, every string leads to a state in a unique way.

For timed languages, it is not obvious how to generalize Nerode's right congruence. In general there is no unique minimal DERA which is equivalent to a given DERA. Consider Figure 1, assuming for a moment the c-transition from location 7 to 5 is missing, then the language of the automaton does not change when changing the transition from 1 to 4 to 1 to 5, although the language accepted from 4 is different then the one from 5. Furthermore, we can reach location 4 by two guarded words: $(b, \mathit{true})(c, x_b \geq 3)$ as well as $(a, \mathit{true})(b, \mathit{true})$. Although they lead to the same state, they admit different continuations of event-clock words: action a can be performed with $x_b = 2$ after $(a, \mathit{true})(b, \mathit{true})$ but not after $(b, \mathit{true})(c, x_b \geq 3)$. The complication is that each past guarded word has a post-condition, which constrains the values of clocks that are possible at the occurrence of future actions.

For a guarded word w_g, we introduce the *strongest postcondition* of w_g, denoted by $sp(w_g)$, as the constraint on clock values that are induced by w_g on any following occurrence of a symbol. Postcondition computation is central in tools for symbolic verification of timed automata [8, 6], and can be done inductively as follows:

- $sp(\lambda) = \bigwedge_{a,b \in \Sigma} x_a = x_b$,
- $sp(w_g(a, g)) = ((sp(w_g) \wedge g)[x_a \mapsto 0]) \uparrow$,

where for clock constraint φ and clock x,

- $\varphi[x \mapsto 0]$ is the condition $x = 0 \wedge \exists x.\varphi$,
- $\varphi \uparrow$ is the condition $\exists d.\varphi'$, where d ranges over $\mathbb{R}^{\geq 0}$ and where φ' is obtained from φ by replacing each clock y by $y - d$.

Both operations can be expressed as corresponding operations on clock constraints. We will also introduce the K-approximation *Approx* $(\varphi)_K$ of φ as the clock constraint obtained by changing in constraints of the form $x - y \leq c$ the constant c to $-(K+1)$ when $c < -K$ and c to ∞ when $c > K$. For example, $x \leq K + 2$ is changed to $x \leq \infty$ while $x \geq K + 2$ is changed to $x \geq K + 1$.

Let us now define a class of DERAs that admit a natural definition of right congruences.

Definition 2. *A DERA D is* sharply guarded *if for all guarded words $w_g(a, g) \in \mathcal{L}(dfa(D))$, we have that g is satisfiable and*

$$g = \bigwedge \{g' \in G \mid sp(w_g) \wedge g' = sp(w_g) \wedge g\}$$

We remark that whether or not a DERA is sharply guarded depends only on $\mathcal{L}(\mathit{dfa}(D))$. In other words, a DERA is called sharply guarded if whenever a run of $\mathcal{L}(\mathit{dfa}(D))$ has reached a certain location l, then the outgoing transitions from l have guards which cannot be strengthened without changing the timing conditions under which the next symbol will be accepted. This does not mean that these guards are "included" in the postcondition (see also Figure 2), but at least their smallest and biggest corners:

Lemma 1. *If $w_g(a, g) \in \mathcal{L}(\mathit{dfa}(D))$, where D is a sharply guarded DERA, then*

1. *there is a timed word $w_t(a, t) \in \mathcal{L}(D)$ such that* $CW(w_t(a, t)) = (a_1, \gamma_1) \ldots (a_n, \gamma_n)(a, \gamma_g) \models w_g(a, g)$ *and* $\gamma_g \in bc(g)$.
2. *there is a timed word $w_t(a, t) \in \mathcal{L}(D)$ such that* $CW(w_t(a, t)) = (a_1, \gamma_1) \ldots (a_n, \gamma_n)(a, \gamma_g) \models w_g(a, g)$ *and* $\gamma_g = sc(g)$.

Proof. The claim follows easily from the definition of sharply guarded. □

Every DERA can be transformed into an equivalent DERA that is sharply guarded using the zone-graph construction [1].

Lemma 2. *For every DERA there is an equivalent DERA that is sharply guarded.*

Proof. Let the DERA $D = \langle \Sigma, L, l_0, \delta, \eta \rangle$ be K-bounded. We define an equivalent sharply guarded DERA $D' = \langle \Sigma, L', l'_0, \delta', \eta' \rangle$ based on the so-called zone automaton for D. We sketch the construction, details can be found in [1, 7]. The set of locations of D' comprises pairs (l, φ) where $l \in L$ and φ is a K-bounded clock constraint. The intention is that φ is the postcondition of any run from the initial location to (l, φ). For any symbol a such that $\delta(l, a)$ is defined and $\varphi \wedge \eta(l, a)$ is satisfiable, let $\delta'((l, \varphi), a)$ be defined as $(\delta(l, a), \varphi'')$ where $\varphi'' = \mathit{Approx}\,(((\varphi \wedge \eta(l, a))[x_a \mapsto 0]) \uparrow)_K$.We set $\eta'((l, \varphi), a) = g'$ with $g' = \bigwedge \{g'' \mid \varphi \wedge g'' = \varphi \wedge \eta(l, a)\}$. It is routine to show that the part of the automaton reachable from the initial location (l_0, true) is sharply guarded. □

The important property of sharply guarded DERAs is that equivalence coincides with equivalence on the corresponding DFAs.

Definition 3. *We call two sharply guarded DERAs D_1 and D_2* dfa-equivalent, *denoted by $D_1 \equiv_{\mathit{dfa}} D_2$, iff $\mathit{dfa}(D_1)$ and $\mathit{dfa}(D_2)$ accept the same language (in the sense of DFAs).*

Lemma 3. *For two sharply guarded DERAs D_1 and D_2, we have*

$$D_1 \equiv_t D_2 \text{ iff } D_1 \equiv_{\mathit{dfa}} D_2$$

Proof. The direction from right to left follows immediately, since $\mathcal{L}(D_i)$ is defined in terms of $\mathcal{L}(\mathit{dfa}(D_i))$. To prove the other direction, assume that $D_1 \not\equiv_{\mathit{dfa}} D_2$. Then there is a shortest w_g such that $w_g \in \mathcal{L}(\mathit{dfa}(D_1))$ but $w_g \notin \mathcal{L}(\mathit{dfa}(D_2))$ (or the other way around). By Lemma 1 this implies that there is a timed word w_t such that $w_t \in \mathcal{L}(D_1)$ but $w_t \notin \mathcal{L}(D_2)$, i.e., $D_1 \not\equiv_t D_2$. □

We can now prove the central property of sharply guarded DERAs.

Theorem 1. *For every DERA there is a unique equivalent minimal sharply guarded DERA (up to isomorphism).*

Proof. By Lemma 2, each DERA D can be translated into an equivalent DERA D' that is sharply guarded. Let $\mathcal{A}_{min}$ be the unique minimal DFA which is equivalent to $\mathit{dfa}(D')$ (up to isomorphism). Since (as was remarked after Definition 2) whether or not a DERA is sharply guarded depends only on $\mathcal{L}(\mathit{dfa}(D))$, we have that $D_{min} = \mathit{dera}(\mathcal{A}_{min})$ is sharply guarded. By Lemma 3, D_{min} is the unique minimal sharply guarded DERA (up to isomorphism) such that $D_{min} \equiv D'$, i.e., such that $D_{min} \equiv D$. □

4 Learning DERAs

Let us now turn to the problem of learning a timed language $\mathcal{L}(D)$ accepted by a DERA D. In this setting, we assume

- to know an upper bound K on the constants occurring in guards of D,
- to have a *Teacher* who is able to answer two kinds of queries:
  - A *membership query* consists in asking whether a timed word w_t over Σ is in $\mathcal{L}(D)$.
  - An *equivalence query* consists in asking whether a hypothesized DERA H is correct, i.e., whether $\mathcal{L}(H) = \mathcal{L}(D)$. The *Teacher* will answer *yes* if H is correct, or else supply a counterexample u, either in $\mathcal{L}(D) \setminus \mathcal{L}(H)$ or in $\mathcal{L}(H) \setminus \mathcal{L}(D)$.

Based on the observations in the previous section, our solution is to learn $\mathcal{L}(\mathit{dfa}(D))$, which is a regular language and can therefore be learned in principle using Angluin's learning algorithm. However, Angluin's algorithm is designed to query (untimed) words rather than timed words. Let us recall Angluin's learning algorithm, before we present our solution in more detail.

4.1 Learning a DFA

Angluin's learning algorithm is designed for learning a regular (untimed) language, $\mathcal{L}(\mathcal{A}) \subseteq \Gamma^*$, accepted by a minimal deterministic finite automaton (DFA) $\mathcal{A}$ (when adapted to the case that $\mathcal{L}(\mathcal{A})$ is prefix-closed). In this algorithm a so called *Learner*, who initially knows nothing about $\mathcal{A}$, is trying to learn $\mathcal{L}(\mathcal{A})$ by asking queries to a *Teacher*, who knows $\mathcal{A}$. There are two kinds of queries:

- A *membership query* consists in asking whether a string $w \in \Gamma^*$ is in $\mathcal{L}(\mathcal{A})$.
- An *equivalence query* consists in asking whether a hypothesized DFA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{A})$. The *Teacher* will answer *yes* if $\mathcal{H}$ is correct, or else supply a counterexample w, either in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}(\mathcal{A})$.

The *Learner* maintains a prefix-closed set $U \subseteq \Gamma^*$ of prefixes, which are candidates for identifying states, and a suffix-closed set $V \subseteq \Gamma^*$ of suffixes, which are used to distinguish such states. The sets U and V are increased when needed during the algorithm. The *Learner* makes membership queries for all words in $(U \cup U\Gamma)V$, and organizes the results into a *table* T which maps each $u \in (U \cup U\Gamma)$ to a mapping $T(u) : V \mapsto \{\text{accepted}, \text{not accepted}\}$. In [4], each function $T(u)$ is called a *row*. When T is *closed* (meaning that for each $u \in U$, $a \in \Gamma$ there is a $u' \in U$ such that $T(ua) = T(u')$) and *consistent* (meaning that $T(u) = T(u')$ implies $T(ua) = T(u'a)$), then the *Learner* constructs a hypothesized DFA $\mathcal{H} = \langle \Gamma, L, l_0, \delta \rangle$, where $L = \{T(u) \mid u \in U\}$ is the set of distinct rows, l_0 is the row $T(\lambda)$, and δ is defined by $\delta(T(u), a) = T(ua)$, and submits $\mathcal{H}$ in an equivalence query. If the answer is *yes*, the learning procedure is completed, otherwise the returned counterexample is used to extend U and V, and perform subsequent membership queries until arriving at a new hypothesized DFA, etc.

4.2 Learning a sharply guarded DERA

Given a timed language that is accepted by a DERA D, we can assume without loss of generality that D is the unique minimal and sharply guarded DERA that exists due to Theorem 1. Then D is uniquely determined by its symbolic language of $\mathcal{A} = \mathit{dfa}(D)$, which is a regular (word) language. Thus, we can learn $\mathcal{A}$ using Angluin's algorithm and return $\mathit{dera}(\mathcal{A})$. However, $\mathcal{L}(\mathcal{A})$ is a language over guarded words, but the *Teacher* in the timed setting is supposed to deal with timed words rather than guarded words.

Let us therefore extend the *Learner* in Angluin's algorithm by an *Assistant*, whose role is to answer a membership query for a guarded word, posed by the *Learner*, by asking several membership queries for timed words to the (timed) *Teacher*. Furthermore, it also has to answer equivalence queries, consulting the timed *Teacher*.

Learning guarded words To answer a membership query for a guarded word w_g, the *Assistant* first extracts the word w underlying w_g. It thereafter determines the unique guard refinement w'_g of w that is accepted by $\mathcal{A}$ (if one exists) by posing several membership queries to the (timed) *Teacher*, in a way to be described below. Note that each word w has at most one guard refinement accepted by $\mathcal{A}$. Finally, the *Assistant* answers the query by *yes* iff w'_g equals w_g.

The guard refinement of w accepted by $\mathcal{A}$ will be determined inductively, by learning the guard under which an action a is accepted, provided that a sequence u of actions has occurred so far. Letting u range over successively longer prefixes of w, the *Assistant* can then learn the guard refinement w'_g of w. Let $u = a_1 a_2 \ldots a_n$, and assume that for $i = 1, \ldots, n$, the *Assistant* has previously learned the guard $g_i = \eta(a_1 \ldots a_{i-1}, a_i)$ under which a_i is accepted, given that the sequence $a_1 \ldots a_{i-1}$ has occurred so far. He can then easily compute the strongest postcondition $\mathit{sp}((a_1, g_1) \ldots (a_n, g_n)) =: \mathit{sp}(u)$. A typical situation for two clocks is depicted in Figure 2. The *Assistant* must now determine the strongest guard g_a such that a is accepted after u precisely when

$\varphi_a \equiv sp(u) \wedge g_a$ holds. In other words, he establishes the hypercube identified by guarding the polyhedron identified by φ_a in which a is accepted. As before, the constraint φ_a for a depends only on the sequence of symbols in u, not on the timing of their occurrence.

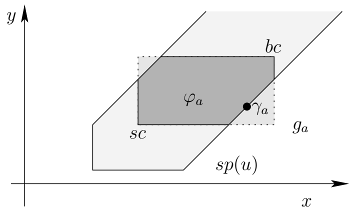


Fig. 2. An example for $sp(u)$, φ_a, and g_a.

The guard g_a is determined by inquiring whether a set of clock valuations γ_a satisfies φ_a. Without loss of generality, the *Assistant* works only with integer valuations. For each γ_a that satisfies the postcondition $sp(u)$, he can make a membership query by constructing a timed word w_t that satisfies the guarded word $(a_1, g_1)(a_2, g_2) \ldots (a_n, g_n)(a, g(\gamma_a))$, where $g(\gamma_a) \equiv \bigwedge\limits_b (x_b = \gamma_a(x_b))$ constrains the clocks to have the values given by γ_a. Note that such a w_t exists precisely when $\gamma_a \models sp(u)$. In other words, he can ask the (timed) *Teacher* for every point in the zone $sp(u)$ whether it is in φ_a (see Figure 2).

Let us now describe how clock valuations γ_a are chosen in membership queries in order to learn the guard g_a for a.

As mentioned before, we assume that the *Assistant* knows the maximal constant K that can appear in any guard. This means that if a clock valuation γ with $\gamma(x) > K$ satisfies g, then clock x has no upper bound in g. Thus, a guard g is uniquely determined by some biggest corner and its smallest corner.

Let us consider how to find a maximal clock valuation that satisfies g_a. Suppose first that the *Assistant* knows some clock valuation γ_a that satisfies φ_a. The *Assistant* will then repeatedly increase the clock values in γ_a until γ_a becomes the maximal clock valuation satisfying g_a. At any point in time, let *Max* be the set of clocks for which *Assistant* knows that they have reached a maximum, and let $\overline{Max} = C_\Sigma \setminus Max$ be the clocks for which a maximum value is still searched. Initially, *Max* is empty and $\overline{Max} = C_\Sigma$. At each iteration, the *Assistant* increases the clocks in $\overline{Max}$ by the same amount $k \in \{1, \ldots, K+1\}$ such that $\gamma_a[\overline{Max} \oplus k] \models \varphi_a$, but $\gamma_a[\overline{Max} \oplus (k+1)] \not\models \varphi_a$, and then sets $\gamma_a := \gamma_a[\overline{Max} \oplus k]$. Here, $\gamma[C \oplus k]$ is defined as $\gamma(x) + k$ for $x \in C$ and $\gamma(x)$ otherwise. This can be done by binary search using at most $\log K$ queries. For all clocks x with $\gamma_a(x) \geq K+1$ he concludes that x has no upper bound in φ_a. These clocks are moved over from $\overline{Max}$ to *Max*. If $\gamma_a(x) \leq K$ for some clock $x \in \overline{Max}$ then among these a clock x must be found that cannot be increased, and this will be moved over from $\overline{Max}$ to *Max*.

Let us examine how to find a clock x that cannot be increased, i.e., for all γ' with $\gamma'(x) > \gamma_a(x)$ we have $\gamma' \not\models \varphi_a$. The particularity to handle is that it might be possible to increase x but only together with other clocks, since $sp(u)$ must be satisfied (e.g., γ_a as in Figure 2 requires both x and y to be incremented to stay in $sp(u)$). We define $d(x) = \bigcap\{C \mid x \in C \text{ and } \gamma_a[C \oplus 1] \models sp(u)\}$ as the clocks dependent on x. In other words, if x is incremented in γ_a so should be the clocks in $d(x)$ since otherwise $sp(u)$ is not satisfied. Note that $x \in d(x)$. Now, he queries for every clock x whether $\gamma_a[d(x) \oplus 1] \models \varphi_a$. If not, he moves x to Max.

This can be optimized in the following way. The dependency graph of $sp(u)$ has nodes $\overline{Max}$ and edges $x \to y$ iff $y \in d(x)$. We define its initial nodes as nodes of strongly connected components that have no incoming edge. He can then use a divide-and-conquer technique to find initial nodes that cannot be incremented using $\log |C_\Sigma|$ queries.

If such a clock x is found then the loop continues and another k is computed. Otherwise, a maximal valuation is found.

Thus, all in all, determining the upper bound of a guard g_a needs at most $|C_\Sigma|$ binary searches, since in every loop at least one clock is moved to Max. Each uses at most $\log K + \log |C_\Sigma|$ membership queries. He can use the same idea to find the minimal clock valuation that satisfies φ_a. g_a is given by the K-approximation of the guard that has the minimal clock valuation as smallest corner and the maximal clock valuation as biggest corner, which can easily be formulated given these two points. Thus, the *Assistant* needs at most $2|C_\Sigma|(\log K + \log |C_\Sigma|)$ membership queries to learn a guard g_a, if initially it knows a valuation which satisfies φ_a.

Suppose now that the *Assistant* does not know a clock valuation γ_a that satisfies φ_a. In principle, φ_a and therefore g_a could specify exactly one valuation, meaning that the *Assistant* essentially might have to ask membership queries for all $\binom{|\Sigma|+K}{|\Sigma|}$ integer points that could be specified by φ_a. This is the number of non-increasing sequences of $|\Sigma| = |C_\Sigma|$ elements, where each element has values among 0 to K, since $sp(u)$ defines at least an ordering on the clocks.

Thus, the *Assistant* can answer a query for a guarded word w_g using at most $|w|\binom{|\Sigma|+K}{|\Sigma|}$ (timed) membership queries.

The final algorithm: To complete the learning algorithm, we have to explain how the *Assistant* can answer equivalence queries to the *Learner*. Given a DFA $\mathcal{H}$, the *Assistant* can ask the (timed) *Teacher*, whether $dera(\mathcal{H}) = D$. If so, the *Assistant* replies *yes* to the *Learner*. If not, the *Teacher* presents a timed word w_t that is in $\mathcal{L}(D)$ but not in $\mathcal{L}(dera(\mathcal{H}))$ (or the other way round). For the word w underlying w_t, we can obtain its guard refinement w_g as described in the previous paragraph. Then w_g is in $\mathcal{L}(dfa(D))$ but not in $\mathcal{L}(\mathcal{H})$ (or the other way round, respectively). Thus, the *Assistant* can answer the equivalence query by w_g in this case.

We call the algorithm outlined in the section LSGDERA.

Complexity For Angluin's algorithm it is known that the number of membership queries can be bounded by $O(kn^2m)$, where n is the number of states, k is the size of the alphabet, and m is the length of the longest counterexample. The rough idea is that for each entry in the table T a query is needed, and $O(knm)$ is the number of rows, n the number of columns.

In our setting, a single membership query for a guarded word w_g might give rise to $|w|\binom{|\Sigma|+K}{|\Sigma|}$ membership queries to the (timed) *Teacher*. While the alphabet of the DFA $dfa(D)$ is $\Sigma \times G$, a careful analysis shows that k can be bounded by $|\Sigma|$ in our setting as well. Thus, the query complexity of LSGDERA for a sharply guarded DERA with n locations is

$$O\left(kn^2ml\binom{|\Sigma|+K}{|\Sigma|}\right)$$

where l is the length of the longest guarded word queried. Since the longest word queried and the longest counterexample can be bounded by $O(n)$, we get at most polynomially many membership queries, in the number of locations as well in the size of the biggest constant K. The number of equivalence queries remains at most n. Note that, in general a (non-sharply guarded) DERA D gives rise to a sharply guarded DERA with double exponentially more locations, while the constants do not change.

4.3 Learning non-sharply guarded DERAs

Learning a sharply guarded DERA allows to transfer Angluin's setting to the timed world. However, in practice, one might be interested in a smaller non-sharply guarded DERA rather than its sharply guarded version. In this section, we describe to learn a usually smaller, non-sharply guarded version. The idea is to identify states whose futures are "similar". While in the worst-case, the same number of membership queries is needed, we can expect the algorithm to converge faster in practice.

Let us now define a relationship on guarded words, which will be used to merge states whose futures are "similar", taking the postcondition into account.

Let $PG = \{\langle\varphi_1, (a_1, g_{11}) \dots (a_n, g_{1n})\rangle, \dots, \langle\varphi_k, (a_1, g_{k1}) \dots (a_n, g_{kn})\rangle\}$ be a set of k pairs of postconditions and guarded words with the same sequences of actions. We say that the guarded word $(a_1, \hat{g_1}) \dots (a_n, \hat{g_n})$ *unifies* PG if for all $j \in \{1, \dots, k\}$ and $i \in \{1, \dots, n\}$

$$g_{ji} \wedge sp(\varphi_j, (a_1, g_{j1}) \dots (a_{i-1}, g_{j(i-1)})) \equiv \hat{g}_i \wedge sp(\varphi_j, (a_1, \hat{g}_1) \dots (a_{i-1}, \hat{g}_{i-1}))$$

Then, the set PG is called *unifiable* and $(a_1, \hat{g_1}) \dots (a_n, \hat{g_n})$ is called a *unifier*. Intuitively, the guarded words with associated postconditions can be unified if there is a unifying, more liberal guarded word, which is equivalent to all guarded words in the context of the respective postconditions. Then, given a set of guarded words with postconditions among $\{\varphi_1, \dots, \varphi_k\}$, these guarded words can be considered to yield the same state, provided that the set of future guarded actions together with the respective postcondition is unifiable.

It is easy to check, whether PG is unifiable, using the property that the guards in the PG are tight in the sense of Definition 2. The basic idea in each step is to take the weakest upper and lower bounds for each variable. Assume the guard g_{ji} is given by its upper and lower bounds:

$$g_{ji} = \bigwedge_{a \in \Sigma} (x_a \leq c^{\leq}_{a,ji} \wedge x_a \geq c^{\geq}_{a,ji})$$

For $i = 1, \ldots, n$, define the candidate $\hat{g}_i$ as

$$\hat{g}_i = \bigwedge_a \left(x_a \leq \max_j \{c^{\leq}_{a,ji}\} \right) \wedge \bigwedge_a \left(x_a \geq \min_j \{c^{\geq}_{a,ji}\} \right)$$

and check whether the guarded word $(a_1, \hat{g}_1) \ldots (a_n, \hat{g}_n)$ obtained in this way is indeed a unifier. It can be shown that if PG is unifiable, then this candidate is the strongest possible unifier.

The learning algorithm using the idea of unified states works similar as the one for DERAs. However, we employ a slightly different observation table. Let $\Gamma = \Sigma \times G$. Rows of the table are guarded words of a prefix-closed set $U \subseteq \Gamma^*$. Column labels are untimed words from a suffix-closed set $V \subseteq \Sigma^*$. The entries of the table are sequences of guards describing under which values the column label extends the row label. Thus, we define a *timed observation table* $T : U \cup U\Gamma \rightarrow (V \rightarrow G^*)$, where $T(u)(v) = g_1 \ldots g_n$ implies $|v| = n$. We require the initial observation table to be defined over $U = \{\lambda\}$ and $V = \Sigma \cup \{\lambda\}$.

A *merging* of the timed observation table T consists of a partition Π of the guarded words $U \cup U\Gamma$, and an assignment of a clock guard $CG(\pi, a)$ to each block $\pi \in \Pi$ and action $a \in \Sigma$, such that for each block $\pi \in \Pi$ we have

- for each suffix $v \in V$, the set $\{\langle sp(u), (a_1, g_1) \ldots (a_n, g_n) \rangle \mid u \in U', T(u)(v) = g_1 \ldots g_n\}$ is unifiable, and
- $(a, CG(\pi, a))$ is the unifier for $\{\langle sp(u), (a, g') \mid u \in \pi, u(a, g') \in U\Gamma\}$ for each $a \in \Sigma$.

Intuitively, a merging defines a grouping of rows into blocks, each of which can potentially be understood as a state in a DERA, together with a choice of clock guard for each action and block, which can be understood as a guard for the action in the DERA. For each table there are in general several possible mergings, but the number of mergings is bounded, since the number of partitions is bounded, and since the number of possible unifiers $GC(\pi, a)$ is also bounded.

A merging Π is *closed* if for every $\pi \in \Pi$ there exists $u \in \Pi$ and $u \in U$, i.e., at least one representative of each block is in the upper part of the table. A merging Π is *consistent* if for all blocks π, whenever $u, u' \in \Pi$, then for all $a \in \Sigma$, for which there there are clock guards g and g' such that $T(u(a, g)) \neq \mathit{false}$ and $T(u'(a, g')) \neq \mathit{false}$, i.e., there is an a-successor, there is a block π' such that $u(a, g) \in \pi'$ and $u'(a, g') \in \pi'$. A *coarsest merging* of the timed observation table T is a merging with a minimal number of blocks.

Given a merging (Π, GC) of a closed and consistent timed observation table T, one can construct the DERA $\mathcal{H} = \langle \Sigma, L, l_0, \delta, \eta \rangle$ as

- $L = \Pi$ comprises the blocks of Π as locations,
- $l_0 = \pi \in \Pi$ with $\lambda \in \pi$ is the initial location,
- δ is defined by $\delta(\pi, a) = \pi'$, where for $u \in \pi$, $u \in U$, $u(a, g) \in \pi'$ and we require $T(u, (a, g)) \neq \mathit{false}$ if such u and g exist.
- η is defined by $\eta(\pi, a) = GC(\pi, a)$.

The algorithm *LDERA* for learning (non-sharply guarded) DERAS is as LSGDERA, except that the new notions of closed and consistent are used. This implies that rows are unified such that the number of blocks is minimal. One further modification is that the hypothesis is constructed as described in the previous paragraph, using the computed merging. The rest of the algorithm remains unchanged.

Lemma 4. *The algorithm LDERA terminates.*

Proof. Assume that we have a machine to learn. We assume a model $\mathcal{A}$ of it that is a sharply guarded DERA. Suppose the algorithm for learning non-sharply guarded DERAs does not terminate. Then it will produce an infinite sequence of closed and consistent observation tables $T_1, \ldots,$, each $T_i : U_i \cup U_i\Gamma \to (V_i \to (G^* \cup \{\text{not accepted}\}))$. Every step of LDERA increases the number of states of the automaton or creates a new automaton with the same number of states, because a (equivalence) query either introduces new states or changes the accepted language. Since the number of different automata with the same number of states is finite, the sequence $T_1, \ldots$ defines a sequence of hypothesis of $\mathcal{A}$ with an increasing number of states.

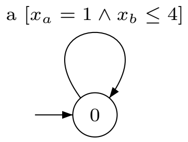
On the other hand, a table T_i can also be understood as a an observation table suitable for the algorithm of learning sharply guarded DERAs: Let V_w be the set of all possible guarded words over $w \in V_i$, and, let $V_i' = \cup_{w \in V_i} V_w$. Such an observation table is coincides all with $\mathcal{A}$ on all strings listed in the table. As such, it can be used as an initial table for LSGDERA, which, would make it closed and consistent as a first step, yielding T_i''.

The automaton corresponding to T_i'' has at least as many states as the automaton that corresponds to the one by table T_i, since in the latter, states are merged. When continuing LSGDERA on T_i'', it will terminate with table T_{r_i}' as the table that corresponds to $\mathcal{A}$. Thus, the automaton corresponding to T_i'' has less states than $\mathcal{A}$. Thus, all automata corresponding to T_i have less states than $\mathcal{A}$. Therefore, the sequence cannot exist. □

Roughly, LDERA can be understood as LSGDERA plus merging. Therefore, in the worst case, more steps and therefore queries are needed as in LSGDERA. However, when a small non-sharply guarded DERA represents a large sharply guarded DERA, LDERA will terminate using less queries. Therefore, a better performance can be expected in practice.

5 Example

In this section, we illustrate the algorithms LDERA and LSGDERA on a small example. Let the automaton A_1 shown in Figure 3(a) be the DERA to learn. We assume that initially clocks x_a and x_b are equal to 0. After a number of queries of the algorithm LDERA, we obtain the observation table T shown in



(a) Automaton A_1

T	λ	a
u_1	true	$x_a = 1 \wedge x_b = 1$
u_2	true	$x_a = 1 \wedge x_b = 2$
u_3	true	$x_a = 1 \wedge x_b = 3$
u_4	true	$x_a = 1 \wedge x_b = 4$
u_5	true	$false$

(b) Table T

Fig. 3. A DERA to learn and an observation table

Figure 3(b), where the guarded words u_1 - u_5 are defined by

$$\begin{aligned}
&u_1 = (\lambda, x_a = 0 \wedge x_b = 0)\\
&u_2 = (a, x_a = 1 \wedge x_b = 1)\\
&u_3 = (a, x_a = 1 \wedge x_b = 1)(a, x_a = 1 \wedge x_b = 2)\\
&u_4 = (a, x_a = 1 \wedge x_b = 1)(a, x_a = 1 \wedge x_b = 2)(a, x_a = 1 \wedge x_b = 3)\\
&u_5 = (a, x_a = 1 \wedge x_b = 1)(a, x_a = 1 \wedge x_b = 2)(a, x_a = 1 \wedge x_b = 3)(a, x_a = 1 \wedge x_b = 4)
\end{aligned}$$

It turns out that all rows of T are unifiable. Define PG by

$$\begin{aligned}
PG = \{\ &\langle sp(u_1), (a, x_a = 1 \wedge x_b = 1)\rangle,\\
&\langle sp(u_2), (a, x_a = 1 \wedge x_b = 2)\rangle,\\
&\langle sp(u_3), (a, x_a = 1 \wedge x_b = 3)\rangle,\\
&\langle sp(u_4), (a, x_a = 1 \wedge x_b = 4)\rangle,\\
&\langle sp(u_5), (a, false)\rangle\}
\end{aligned}$$

It can be checked that the guarded word $(a, x_a = 1 \wedge x_b \leq 4)$ unifies PG. We will use the merging of the observation table T as the partition which consists of exactly one block, and equipping the action a with the guard $x_a = 1 \wedge x_b \leq 4$. The automaton obtained from this mergings is the automaton A_1 which consists of exactly one state. In contrast, the algorithm LSGDERA, which does not employ unification, would construct the sharply guarded DERA A_2 shown in Figure 4. The automaton A_2 has 5 states, since table T has 5 different rows.

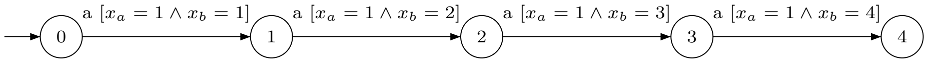

a [$x_a = 1 \wedge x_b = 1$]
a [$x_a = 1 \wedge x_b = 2$]
a [$x_a = 1 \wedge x_b = 3$]
a [$x_a = 1 \wedge x_b = 4$]
0
1
2
3
4

Fig. 4. Automaton A_2

6 Conclusion

In this paper, we presented a technique for learning timed systems that can be represented as event-recording automata. By considering the restricted class of event-deterministic automata, we can uniquely represent the automaton by a regular language of guarded words, and the learning algorithm can identify states by access strings that are untimed sequences of actions. This allows us to adapt existing algorithms for learning regular languages to the timed setting. The main additional work is to learn the guards under which individual actions will be accepted. Without the restriction of event-determinism, learning becomes significantly less tractable, since we must also learn timing constraints of past actions under which guards on current actions are relevant. This might be possible in principle, representing the language by a regular language of guarded words, e.g., as in [22], but would lead to an explosion in the number of possible access strings.

The complexity of our learning algorithm is polynomial in the size of the minimal zone graph. In general, this can be doubly exponentially larger than a minimal DERA automaton representing the same language, but for many practical systems the zone graph construction does not lead to a severe explosion, as exploited by tools for timed automata verification [8, 6]. Furthermore, we discussed learning of not necessarily sharply guarded DERAs directly to quickly obtain smaller representations of the system to learn. It would be interesting to establish lower bounds of the learning problem for timed systems.

Acknowledgments. We thank Paul Pettersson and Rafał Somla for insightful discussions, and Oded Maler for sending his paper [22].

References

1. R. Alur. Timed automata. In *Proc. 11th International Computer Aided Verification Conference*, LNCS 1633, p. 8–22. Springer, 1999.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. R. Alur, L. Fix, and T. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211:253–273, 1999.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
5. J. L. Balcázar, J. Díaz, and R. Gavaldá. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, p. 53–72. Kluwer, 1997.
6. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, p. 232–243. Springer, 1996.
7. P. Bouyer. Untameable timed automata. In H. Alt and M. Habib, editors, *Symp. on Theoretical Aspects of Computer Science*, LNCS 2607. Springer, 2003.
8. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, LNCS 1427, p. 546–550. Springer, 1998.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Dec. 1999.

10. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. 22nd Int. Conf. on Software Engineering*, June 2000.
11. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite-State Systems*, LNCS 407. Springer, 1989.
12. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.
13. E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.
14. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280. Springer, 2002.
15. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02,* 5^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, LNCS 2306, p. 80–95. Springer, 2002.
16. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403–414, April 1990.
17. T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112:173–337, 1994.
18. T. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In K. Larsen, S. Skuym, and G. Winskel, editors, *Proc. ICALP '98,* 25^{th} *International Colloquium on Automata, Lnaguages, and Programming*, LNCS 1443, p. 580–591. Springer, 1998.
19. G. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN Model Checking and Software Verification: Proc.* 7^{th} *Int. SPIN Workshop*, LNCS 1885, p. 131–147, Stanford, CA, 2000. Springer.
20. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003.
21. M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.
22. O. Maler and A. Pnueli. On recognizable timed languages. In *Proc. FOSSACS04, Conf. on Foundations of Software Science and Computation Structures*, LNCS. Springer, 2004. Available from `http://www-verimag.imag.fr/PEOPLE/Oded.Maler/`.
23. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
24. M. Schmitt, M. Ebner, and J. Grabowski. Test generation with Autolink and Testcomposer. In *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'2000*, June 2000.
25. J. Springintveld and F. Vaandrager. Minimizable timed automata. In B. Jonsson and J. Parrow, editors, *Proc. FTRTFT'96, Formal Techniques in Real-Time and Fault-Tolerant Systems, Uppsala, Sweden*, LNCS 1135, p. 130–147. Springer, 1996.
26. T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Proc. FTRTFT'94, Formal Techniques in Real-Time and Fault-Tolerant Systems, Lübeck, Germany*, LNCS 863, p. 694–715. Springer, 1994.