Relating timed and register automata *

Diego Figueira
INRIA, ENS Cachan, LSV
France

Piotr Hofman
Institute of Informatics
University of Warsaw
Poland

Sławomir Lasota
Institute of Informatics
University of Warsaw
Poland

Timed automata and register automata are well-known models of computation over timed and data words respectively. The former has *clocks* that allow to test the lapse of time between two events, whilst the latter includes *registers* that can store data values for later comparison. Although these two models behave in appearance differently, several decision problems have the same (un)decidability and complexity results for both models. As a prominent example, emptiness is decidable for alternating automata with one clock or register, both with non-primitive recursive complexity. This is not by chance.

This work confirms that there is indeed a tight relationship between the two models. We show that a run of a timed automaton can be simulated by a register automaton, and conversely that a run of a register automaton can be simulated by a timed automaton. Our results allow to transfer complexity and decidability results back and forth between these two kinds of models. We justify the usefulness of these reductions by obtaining new results on register automata.

1 Introduction

Timed automata [2] and register automata (known originally as *finite-memory automata*) [8] are two widely studied models of computation. Both models extend finite automata with a kind of storage: *clocks* in the case of timed automata, capable of measuring the amount of time elapsed from the moment they were reset; and *registers* in the case of register automata, capable of storing a data value for future comparison. In this paper we are interested in decidability and complexity of standard decision problems for both models of automata. In particular, we focus on the problems of *nonemptiness* (Does an automaton $\mathscr{A}$ accept some word?), *universality* (Does an automaton $\mathscr{A}$ accept all words?), and *inclusion* (Are all words accepted by an automaton $\mathscr{A}$ also accepted by an automaton $\mathscr{B}$?).

The emptiness problem for nondeterministic timed or register automata is PSPACE-complete [2, 4]. It becomes undecidable for *alternating* automata of both kinds [9, 15, 4], as soon as they have at least two clocks or registers [2, 4]. Even the universality problem was shown undecidable for nondeterministic timed and register automata, respectively, with two clocks or registers [2, 13, 4]. A break-through result of [14] showed that universality becomes decidable for one clock timed automata. Later, the emptiness problem for one clock alternating timed automata was shown decidable. However, the computational complexity of this problem has been found to be non-primitive recursive [9, 15]. Analogous (independent) results appeared for the other model: emptiness is decidable and non-primitive recursive for one register alternating automata [4]. For infinite words, both one clock and one register alternating automata are undecidable, as well as the universality problem of nondeterministic one clock/register automata [9, 1, 4]. The analogies between the two models appear to some extent also at the level of proof methods. The decidability proofs for one clock/register alternating automata are based on similar

*Work supported by the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

To appear in EPTCS.

well-structured transition systems; and both non-primitive recursive lower bounds are obtained by simulation of a kind of lossy model of computation. All these analogies between the two models rise a natural question about the relationship between them. This paper is an attempt to answer this question.

Register automata were traditionally investigated over an unordered data domain. However, our model works on a data domain equipped with a total order. This is a necessary extension, that allows to simulate runs of timed automata, and to have a tight equivalence between the timed and the register models. Roughly speaking, the main contribution of this paper is to show that

$$\text{timed automata} \approx \text{register automata over an ordered data domain.}$$

On a more technical level, we show that a run of a timed/register automaton on a timed/data word w may be simulated by a run of a register/timed automaton over a specially instrumented transformation of w, that we call *braid*. The reductions we exhibit are performed in exponential time, and keep the number of clocks equal to the number of registers, and preserve the mode of computation (alternating, nondeterministic, deterministic). Additionally, we show that the complement of all braids is recognizable by a nondeterministic one clock/register automaton. These results lead straightforwardly to reductions from decision problems for one class of automata to analogous problems for the other class, thus allowing us to carry over (un)decidability results and derive complexity bounds in both directions.

As an application, our simulations allow to obtain known results on timed (or register) models as simple consequences of results on register (or timed) models. These include, e.g., that over finite words the emptiness problem of alternating 1 register automata is decidable [4]. In fact, our reductions yield decidability of the model extended with a total order over the data domain. As two further examples of application, we show how the following decidability results for timed automata can be transferred to the class of register automata:

- decidability of the inclusion problem between a nondeterministic (many clocks) automaton and an alternating one clock automaton (shown in [9]);
- decidability of the emptiness problem for an alternating (many clocks) automaton over a *bounded* time domain (shown in [7]).

In this paper we limit our study to *finite* timed and data words, as the first step in the general program of relating the timed and data settings.

2 Preliminaries

$\mathbb{R}_+$ denotes the set of non-negative real numbers. Let $\mathscr{B}^+(X)$ denote the set of all positive boolean formulas over the set X of propositions, i.e., the set generated by:

$$\phi \quad ::= \quad x \;\mid\; \phi_1 \wedge \phi_2 \;\mid\; \phi_1 \vee \phi_2 \qquad (x \in X).$$

We fix a finite alphabet $\mathbb{A}$ for the sequel. We recall the definitions of alternating timed and register automata [9, 4]. To avoid inessential technical complications, we have deliberately chosen a slightly unusual definition of register automata, equivalent in terms of expressible power to the one defined in [4], but as similar as possible to the definition of timed automata.

2.1 Alternating timed automata

By a *timed word* over $\mathbb{A}$ we mean a finite sequence

$$w \;=\; (a_1,t_1)\,(a_2,t_2)\,\ldots\,(a_n,t_n) \tag{1}$$

of pairs from $\mathbb{A} \times \mathbb{R}_+$, with $t_1 < t_2 < \ldots < t_n$. Each *time stamp* t_i denotes the amount of time elapsed since the beginning of the word. For simplicity, we prefer to work with strictly monotonic timed words, although the analogous results would hold for weakly monotonic words as well.

For a given finite set $\mathscr{C}$ of *clock variables* (or *clocks* for short), consider the set $\mathtt{Constr}(\mathscr{C})$ of clock constraints σ defined by

$$\sigma \quad ::= \quad c < k \;\mid\; c \leq k \;\mid\; \sigma_1 \wedge \sigma_2 \;\mid\; \neg\sigma,$$

where k stands for an arbitrary nonnegative integer constant, and $c \in \mathscr{C}$. For instance, note that *tt* (standing for *always true*), or $c = k$, can be defined as abbreviations. Recall also that the difference constraints $c_1 - c_2 \leq k$, typically allowed in timed automata, may be easily eliminated (however, the size of automaton may increase exponentially). A *valuation* of the clocks is an element $\mathbf{v} \in (\mathbb{R}_+)^{\mathscr{C}}$. Given a constraint σ we write $[\sigma]$ to denote the set of clock valuations satisfying the constraint, $[\sigma] \subseteq (\mathbb{R}_+)^{\mathscr{C}}$.

An *alternating timed automaton* over $\mathbb{A}$ consists of: a finite set of states Q, a distinguished initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, a finite set $\mathscr{C}$ of clocks, and a finite partial transition function

$$\delta : Q \times \mathbb{A} \times \mathtt{Constr}(\mathscr{C}) \dot{\to} \mathscr{B}^+(Q \times \mathscr{P}(\mathscr{C})),$$

subject to the following additional restriction:

(Partition) For every state q and label a, $\{[\sigma] : \delta(q,a,\sigma) \text{ is defined}\}$ is a (finite) partition of $(\mathbb{R}_+)^{\mathscr{C}}$.

The (Partition) condition does not limit the expressive power of automata. We impose it because it permits to give a nice symmetric semantics for the automata as explained below. We will write $q, a, \sigma \;\mapsto\; b$ instead of $\delta(q,a,\sigma) = b$.

To define an execution of an automaton, we will need two operations on valuations $\mathbf{v} \in (\mathbb{R}_+)^{\mathscr{C}}$. A valuation $\mathbf{v}{+}t$, for $t \in \mathbb{R}_+$, is obtained from $\mathbf{v}$ by increasing the value of each clock by t. A valuation $\mathbf{v}[X := 0]$, for $X \subseteq \mathscr{C}$, is obtained by reseting to zero the value of all clocks from X.

For an alternating timed automaton $\mathscr{A}$ and a timed word w as in (1), we define the *acceptance game* $G^{\text{time}}_{\mathscr{A},w}$ between two players Adam and Eve. Intuitively, the objective of Eve is to accept w, while the aim of Adam is the opposite. A play starts at the initial configuration $(q_0, \mathbf{v}_0)$, where $\mathbf{v}_0 : \mathscr{C} \to \mathbb{R}_+$ is a valuation assigning 0 to each clock variable. It consists of n phases. The $(k{+}1)$-th phase starts in $(q_k, \mathbf{v}_k)$, and ends in some configuration $(q_{k+1}, \mathbf{v}_{k+1})$ proceeding as follows. Let $\bar{\mathbf{v}} := \mathbf{v}_k + t_{k+1} - t_k$ (for $k = 0$, t_0 is deemed to be 0). Let σ be the unique constraint such that $\bar{\mathbf{v}}$ satisfies σ and $\phi = \delta(q_k, a_{k+1}, \sigma)$ is defined. Existence and uniqueness of such σ is implied by the (Partition) condition. Now the outcome of the phase is determined by the formula ϕ. There are three cases:

- $\phi = \phi_1 \wedge \phi_2$: Adam chooses one of subformulas ϕ_1, ϕ_2 and the play continues with ϕ replaced by the chosen subformula;
- $\phi = \phi_1 \vee \phi_2$: dually, Eve chooses one of subformulas;
- $\phi = (q, X) \in Q \times \mathscr{P}(\mathscr{C})$: the phase ends with the result $(q_{k+1}, \mathbf{v}_{k+1}) := (q, \bar{\mathbf{v}}[X := 0])$. A new phase is starting from this configuration if $k{+}1 < n$.

The winner is Eve if q_n is accepting ($q_n \in F$), otherwise Adam wins.

Formally, a play is a finite sequence of consecutive game positions of the form $\langle k, q, \mathbf{v}\rangle$ or $\langle k, q, \phi\rangle$, where k is the phase number, ϕ a positive boolean formula, q a state and $\mathbf{v}$ a valuation. A *strategy* of Eve is a mapping which assigns to each such sequence ending in Eve's position a next move of Eve. A strategy is *winning* if Eve wins whenever she applies this strategy.

The automaton $\mathscr{A}$ *accepts* w iff Eve has a winning strategy in the game $G^{\text{time}}_{\mathscr{A},w}$. By $\mathscr{L}(\mathscr{A})$ we denote the language of all timed words w accepted by $\mathscr{A}$.

2.2 Alternating register automata

Fix an infinite data domain $\mathbb{D}$. *Data words* over $\mathbb{A}$ are finite sequences

$$w \;=\; (a_1,d_1)(a_2,d_2)\dots(a_n,d_n) \tag{2}$$

of pairs from $\mathbb{A}\times\mathbb{D}$. Additionally, assume a total order $\preceq$ over $\mathbb{D}$. The order may be chosen arbitrarily, and our results apply to all total orders.

For a given finite set $\mathscr{R}$ of *register names* (or *registers* for short), consider the set $\texttt{Tests}(\mathscr{R})$ of register tests σ defined by

$$\sigma \quad ::= \quad \prec r \;\mid\; \preceq r \;\mid\; \sigma_1\wedge\sigma_2 \;\mid\; \neg\sigma, \qquad \text{where } r\in\mathscr{R}.$$

Each test σ refers to registers and the current data, thus σ denotes a subset $[\sigma]$ of $\mathbb{D}^{\mathscr{R}}\times\mathbb{D}$. E.g., $[\prec r]$ means that the current data value is strictly smaller than the value stored in register r. The equality '$=r$' and inequality '$\neq r$' tests may be defined as abbreviations.

An *alternating register automaton* over $\mathbb{A}$ consists of: a finite set Q of states with a distinguished initial state $q_0\in Q$ and a set of accepting states $F\subseteq Q$, a finite set $\mathscr{R}$ of registers, and a transition function

$$\delta : Q\times\mathbb{A}\times\texttt{Tests}(\mathscr{R}) \to \mathscr{B}^+(Q\times\mathscr{P}(\mathscr{R}))$$

subject to the following additional restriction:

(Partition) For every q and a, the set $\{[\sigma] : \delta(q,a,\sigma) \text{ is defined}\}$ gives a (finite) partition of $\mathbb{D}^{\mathscr{R}}\times\mathbb{D}$.

Register automata are typically defined over unordered data domain. For the purpose of relating the existing models, distinguish a subclass of register automata that only use equality $=_r$ and inequality $\neq_r$ tests; we call them *order-blind automata*. Order-blind automata correspond to the model defined in [4]. As usual, we will write $q,a,t \;\mapsto\; \phi$ instead of $\delta(q,a,t)=\phi$.

Given a data word w as in (2), it is accepted or not by $\mathscr{A}$ depending on the winner in the *acceptance game* $G^{\text{data}}_{\mathscr{A},w}$, played by Eve and Adam similarly as for timed automata. We assume for convenience that as the very first step the automaton loads the current data into all registers (in this way we avoid undefined values in registers). The initial configuration is thus $(q_0,\mathbf{v}_0)$, where $\mathbf{v}_0 : \mathscr{R}\to\mathbb{D}$ assigns d_1 to each register. The play consists of n phases. The $(k{+}1)$-th phase starts in $(q_k,\mathbf{v}_k)$ and proceeds as follows. Let σ be the unique test such that $\mathbf{v}_k$ satisfies σ and $\phi=\delta(q_k,a_{k+1},\sigma)$ is defined (recall the (Partition) condition). Now the outcome of the phase is determined by the formula ϕ. The logical connectives are dealt with analogously as in case of timed automata. When the play reaches an atomic formula $\phi=(q,X)\in Q\times\mathscr{P}(\mathscr{R})$, the phase ends with the result $(q_{k+1},\mathbf{v}_{k+1}) := (q,\mathbf{v}_k[X:=d_{k+1}])$, where $\mathbf{v}[X:=d]$ differs from $\mathbf{v}$ by setting $\mathbf{v}(r)=d$ for all $r\in X$. If $k{+}1<n$, the game continues with a new phase starting in $(q_{k+1},\mathbf{v}_{k+1})$.

The winner is Eve if q_n is accepting ($q_n\in F$), otherwise Adam wins. The automaton $\mathscr{A}$ *accepts* w iff Eve has a winning strategy in $G^{\text{data}}_{\mathscr{A},w}$. Overloading the notation, $\mathscr{L}(\mathscr{A})$ denotes the language of all data words accepted by $\mathscr{A}$.

Deterministic, nondeterministic, and alternating. For both timed and register automata, we distinguish a subclass of nondeterministic automata as those that do not use conjunction in the image of transition function, and a subclass of deterministic automata that do not use disjunction either. The term alternating automata refers then to the full, unrestricted class.

2.3 Isomorphisms

By a *time isomorphism* we mean any order-preserving bijection f over the interval $[0,1)$ (this implies $f(0)=0$ in particular). The intuition is that an isomorphism will not be applied to a time stamp t, but to its fractional part only (that we write $\widehat{t}$), keeping the integer part $\lfloor t \rfloor$ unchanged.

Given a time isomorphism f, we apply it to a timed word $w=(a_1,t_1)\cdots(a_n,t_n)$ as follows:

$$f(w)=(a_1,\lfloor t_1\rfloor+f(\widehat{t_1}))(a_2,\lfloor t_2\rfloor+f(\widehat{t_2}))\cdots(a_n,\lfloor t_n\rfloor+f(\widehat{t_n}))$$

Proposition 2.1. *Languages recognized by alternating timed automata are closed under time isomorphism: for any timed automaton $\mathscr{A}$ and a time isomorphism f, $\mathscr{A}$ accepts a timed word w iff $\mathscr{A}$ accepts $f(w)$.*

We say that two data words $w=(a_1,d_1)(a_2,d_2)\ldots(a_n,d_n)$ and $v=(a_1,e_1)(a_2,e_2)\ldots(a_n,e_n)$ with the same string projection $a_1a_2\ldots a_n$ are *data isomorphic* if for all $i,j\in\{1\ldots n\}$, $d_i\preceq d_j$ iff $e_i\preceq e_j$.

Proposition 2.2. *Languages recognized by alternating register automata are closed under data isomorphism: for any register automaton $\mathscr{A}$ and a two data isomorphic words w and v, $\mathscr{A}$ accepts w iff $\mathscr{A}$ accepts v.*

3 Braids

An idea which is crucial to obtain reductions in both directions is an instrumentation of timed and data words, to be defined in this section, that enforces a kind of 'braid' structure in a word.

Data braids. The *data projection* of $w=(a_1,d_1)\ldots(a_n,d_n)\in(\mathbb{A}\times\mathbb{D})^*$ is $d_1\ldots d_n\in\mathbb{D}^*$. We define the *ordered partition* of a data word w as a factorization

$$w_1\cdot\ldots\cdot w_k=w \tag{3}$$

into data words $w_1,\ldots,w_k$ such that each w_i is a maximal subword ordered with respect to $\prec$. In other words: all the data values of any w_i are strictly increasing, and for all $i<k$, the first data value of w_{i+1} is less or equal to the last one of w_i. It follows that for every data word there is a unique ordered partition.

A data word w is a *data braid* iff

- The minimum data value of w appears at the first position, and
- Its ordered partition is such that the data projection of each factor w_i is a substring of the data projection of w_{i+1}. In this context, we say that v is a *substring* of v' iff v is the result of removing some (possibly none) positions from v'.
- We can partition the alphabet $\mathbb{A}=\mathbb{A}_1\cup\mathbb{A}_2$ so that a position i of w is labeled with a symbol of $\mathbb{A}_2$ iff $d_i=d_1$. We call a *marked position* to any $\mathbb{A}_2$-labeled position of the word. Note that the marked positions are those starting some factor of the ordered partition of w.

Example 3.1. The word w below is not an ordered data braid since its ordered partition does not satisfy the substring requirement. Neither is v, since the minimum element does not appear at the first position. In this example as well as in the following ones we use natural number as exemplary data value.

$$\begin{array}{rcl} w & = & (c,1)\ \cdot\ (d,1)(a,4)(b,8)\ \cdot\ (c,1)(b,2)(a,4)(a,8)(b,9)\ \cdot\ (c,1),\\ v & = & (c,3)\ \cdot\ (d,2)(a,3)(b,8)\ \cdot\ (c,2)(b,3)(a,5)(a,8).\end{array}$$

In the case of w, the substring requirement is fulfilled if, e.g., the last element $(c,1)$ is removed, or when w is extended with $(b,2)(a,4)(b,5)(a,8)(b,9)$; in both cases $\mathbb{A}_1=\{a,b\}$ and $\mathbb{A}_2=\{c,d\}$.

Timed braids. Intuitively, the braid condition for timed words is analogous to that of ordered data braids if one considers the fractional part of a time stamp t_i as datum. A timed word

$$w = (a_1,t_1)(a_2,t_2)\dots(a_n,t_n)$$

is a *timed braid* if the very first time stamp equals zero, $t_1 = 0$, and moreover

- for all $i < n$, if $t_i < \lfloor t_n \rfloor$ then $t_i + 1$ appears among $t_{i+1} \dots t_n$,
- the alphabet can be partitioned into $\mathbb{A} = \mathbb{A}_1 \cup \mathbb{A}_2$ so that the marked positions (i.e., those labeled by $\mathbb{A}_2$) are precisely those carrying *integer* time stamp.

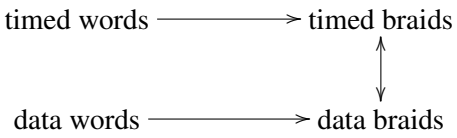
Braids will play a central role in the following section. In fact both data braids and timed braids represent essentially the same concept, disregarding some minor details, as illustrated next.

Example 3.2. We show a data braid w and a 'corresponding' timed braid v. $\mathbb{A}_1 = \{a,b\}$ and $\mathbb{A}_2 = \{\bar{a},\bar{b}\}$.

$$\begin{aligned} w &= (\bar{b},2)(a,4) \;\cdot\; (\bar{a},2)(b,4)(b,8) \;\cdot\; (\bar{b},2)(b,3)(a,4)(a,8)(b,9) \\ v &= (\bar{b},0.0)(a,0.5) \;\cdot\; (\bar{a},1.0)(b,1.5)(b,1.6) \;\cdot\; (\bar{b},2.0)(b,2.3)(a,2.5)(a,2.6)(b,2.9). \end{aligned}$$

The particular data values and time stamps are exemplary ones. A canonical way of obtaining a timed braid from a data braid (and vice versa), to be explained below, will be ambiguous up to time (data) isomorphism.

Transformations. We introduce two simple encodings: one maps a timed word into a data braid, and the other maps a data word into a timed braid.



A timed word w over an alphabet $\mathbb{A}$ induces a timed braid $\mathrm{tb}(w)$ over the extended alphabet $\mathbb{A} \cup \{\checkmark\} \cup \bar{\mathbb{A}} \cup \{\bar{\checkmark}\}$, where $\bar{\mathbb{A}} = \{\bar{a} \mid a \in \mathbb{A}\}$, as follows. First, if $t_1 \neq 0$, add the pair $(\checkmark,0)$ at the very first position. Then add pairs $(\checkmark,t)$ at all time points t that are missing according to the definition of timed braid. Finally change every symbol a at each position carrying an integer time stamp by its 'marked' counterpart $\bar{a} \in \bar{\mathbb{A}} \cup \{\bar{\checkmark}\}$.

A data word w over $\mathbb{A}$ may be canonically extended to a data braid $\mathrm{db}(w)$ over the alphabet $\mathbb{A} \cup \{\checkmark\} \cup \bar{\mathbb{A}} \cup \{\bar{\checkmark}\}$ as follows. Consider the ordered partition $w = w_1 \cdot \ldots \cdot w_n$ and let $d_{\min}$ be the smallest datum appearing in w. Firstly, for every factor w_i, add the pair $(\checkmark, d_{\min})$ at the very first position of w_i, unless w_i already contains the datum $d_{\min}$. Secondly, for each datum d appearing in any w_i, add $(\checkmark,d)$ to each of the following factors $w_{i+1} \dots w_n$ that do not contain d. This insertion is done preserving the order of the factor. Finally, change every symbol a at the first position of a factor by its 'marked' counterpart $\bar{a} \in \bar{\mathbb{A}} \cup \{\bar{\checkmark}\}$. Note that as a result we obtain a data braid.

Example 3.3. As an illustration, consider the effect of the above transformations on an exemplary data word w and a timed word v.

$$\begin{aligned} w &= (a,4) \;\cdot\; (b,1)(a,4)(b,8) \;\cdot\; (a,1)(a,5)(a,8) \\ \mathrm{db}(w) &= (\bar{\checkmark},1)(a,4) \;\cdot\; (\bar{b},1)(a,4)(b,8) \;\cdot\; (\bar{a},1)(\checkmark,4)(a,5)(a,8) \\ v &= (a,0.0)(a,0.7) \;\cdot\; (b,1.5) \;\cdot\; (b,2.0) \\ \mathrm{tb}(v) &= (\bar{a},0.0)(a,0.7) \;\cdot\; (\bar{\checkmark},1.0)(b,1.5)(\checkmark,1.7) \;\cdot\; (\bar{b},2.0)(\checkmark,2.5)(\checkmark,2.7) \end{aligned}$$

We have thus explained the horizontal arrows of the diagram, and now we move to the vertical ones. Both mappings preserve the length of the word.

A timed braid $(a_1,t_1)\dots(a_n,t_n)$ gives naturally rise to a data braid by replacing each time stamp t_i by its fractional part $\widehat{t_i}$, and then mapping the set $\{\widehat{t_1},\dots,\widehat{t_n}\}$ into the data domain $\mathbb{D}$ through an order-preserving injection. We only want to consider order-preserving injections, thus this always yields a data braid. Note that the choice of a particular order-preserving injection is irrelevant, as one always obtains the same data word up to data isomorphism (cf. Proposition 2.2). We hope this ambiguity will not be confusing.

A data braid $w = (a_1,d_1)\cdots(a_n,d_n)$ may be turned into a timed braid through any order-preserving injection $f : \{d_1,\dots,d_n\} \to [0,1)$ such that $f(d_1) = 0$. Each element (a_i,d_i) is mapped into a similar element $(a_i,k+f(d_i))$, where k is the number of factors (in the ordered partition of w) that end strictly before position i. Consecutive factors will get consecutive natural numbers as the integer part of time stamps. As before, we consider the choice of a particular injection f irrelevant (cf. Proposition 2.1).

Notice that going from a timed braid to a data braid and back returns to the original word up to time isomorphism; likewise, combining the transformations in the reverse order we get back to the same word, up to data isomorphism.

Slightly overloading the notation, we write $\mathrm{db}(w)$ to denote the data braid obtained from a *timed* word w by the appropriate composition of transformations just described. Similarly, we write $\mathrm{tb}(w)$ to denote the timed braid obtained from a *data* word w.

4 From timed automata to register automata

We are going to show that, up to a suitable encoding, languages recognized by timed automata are recognized by register automata as well. The transformation keeps the number of registers equal to the number of clocks, and preserve the mode of computation (nondeterministic, alternating).

Theorem 4.1. *Given an alternating timed automaton $\mathscr{A}$ one can compute in exponential time an order-blind register automaton $\mathscr{B}$ such that for any timed word w, $\mathscr{A}$ accepts w if an only if $\mathscr{B}$ accepts $db(w)$. The number of registers of $\mathscr{B}$ equals the number of clocks of $\mathscr{A}$. Moreover, $\mathscr{B}$ is deterministic (resp. nondeterministic, alternating) if $\mathscr{A}$ is so.*

Proof. We describe the construction of a register automaton $\mathscr{B}$ that faithfully simulates a given timed automaton $\mathscr{A}$. The idea is that the behavior of each clock can be simulated by a register. When the clock is reset on one automaton, the other loads the current data value d into the register. Then, by the data braid structure, the register automaton knows exactly how many units of time have elapsed for the clock by simply counting the number of times that d has appeared.

Consider the maximum constant $k_{\max}$ that appears in the transition rules of $\mathscr{A}$. Let Q and $\mathscr{C}$ denote the states and clocks of $\mathscr{A}$, respectively. The states of $\mathscr{B}$ will be $Q \times \{0,1,\dots,k_{\max}\}^{\mathscr{C}}$. Intuitively, for each clock c the automaton $\mathscr{B}$ stores the information about the integer part $\lfloor c \rfloor$ of current value of c, up to $k_{\max}$. In other words, $\mathscr{B}$ keeps the count of how many times (up to $k_{\max}$) the value stored in c appeared in the word since it was stored. The initial state is (q_0,v) where v assigns 0 to each $c \in \mathscr{C}$. Recall that it is assumed that as the very first step the automaton loads the current data into all registers.

There will be as many registers in $\mathscr{B}$ as clocks in $\mathscr{A}$, $\mathscr{R} = \mathscr{C}$, and each register c will be used to update the information about the integer part of the value of c. Whenever the clock c is *reset* in $\mathscr{A}$, the corresponding action of $\mathscr{B}$ is to *store* the current value in the register c and to change state from (q,v) to (q,v_c), where v_c differs from v by assigning $v_c(c) = 0$. The automaton $\mathscr{B}$ will also be capable to detect

that the integer part of c increases. Whenever the equality test $=_c$ succeeds (recall that $\mathscr{B}$ is supposed to run over a data braid) and $v(c) < k_{\max}$, the state is changed from (q,v) to (q,v^c), where v^c differs from v by assigning $v^c(c) = v(c)+1$. On the other hand, v is not changed when $v(c) = k_{\max}$.

Now we describe the transitions of $\mathscr{B}$ in more detail. The automaton does not distinguish marked symbols from unmarked ones. If the current letter is $\checkmark$ or $\bar{\checkmark}$, then $\mathscr{B}$ only needs to update the v part of its state (q,v). Note that in this model many registers may store the same data value, and then there will be a transition in $\mathscr{B}$ for each vector $v \in \{0,1,\ldots,k_{\max}\}^{\mathscr{C}}$ and subset $X \subseteq \mathscr{C}$:

$$(q,v), a, \big(\textstyle\bigwedge_{c\in X} =_c \ \wedge \bigwedge_{c\notin X} \neq_c\big) \ \mapsto \ ((q,v^X),X) \qquad \text{for } a \in \{\checkmark,\bar{\checkmark}\}. \tag{4}$$

where v^X is defined by:

$$v^X(c) = \begin{cases} v(c)+1 & \text{if } c \in X \text{ and } v(c) < k_{\max} \\ v(c) & \text{otherwise.} \end{cases}$$

Otherwise, when the current letter a is different from $\checkmark, \bar{\checkmark}$, the automaton $\mathscr{B}$ simultaneously updates v similarly as above and simulates an actual step of $\mathscr{A}$. Consider any transition

$$q, a, \sigma \ \mapsto \ \phi \tag{5}$$

of $\mathscr{A}$. We describe the corresponding transitions of $\mathscr{B}$. There are many of them, each of them induced by v and X similarly as above. They are of the following form:

$$(q,v), a, \big(\bigwedge_{c\in X} =_c \ \wedge \bigwedge_{c\notin X} \neq_c\big) \ \mapsto \ \phi_v^X \qquad \text{for } a \in \mathbb{A} \cup \bar{\mathbb{A}}, \tag{6}$$

where ϕ_v^X is appropriately obtained from ϕ to ensure that the set of clocks reset by $\mathscr{A}$ is the same as the set of registers to which $\mathscr{B}$ loads the current value, and that the new vector v keeps up-to-date information about the integer parts of clock values. Let us describe how to build ϕ_v^X more precisely.

Consider any fixed vector $v \in \{0,1,\ldots,k_{\max}\}^{\mathscr{C}}$ and $X \subseteq \mathscr{C}$. Being in state (q,v) and reading a next letter a, the automaton assumes that the clock c has a value in $(v(c), v(c)+1]$ if $v(c) < k_{\max}$, or in $(k_{\max},\infty)$ if $v(c) = k_{\max}$. Further, if a clock c verifies $c \in X$, it means that the value of the corresponding register c equals to the current datum. This translates in an *integer* number of units of time elapsed for the clock c, and the exact number (up to $k_{\max}$) is given by $v(c)+1$. The pair $[v,X]$ induces a subset of $(\mathbb{R}_+)^{\mathscr{C}}$ (keep in mind that the test in (6) holds) containing all vectors z such that for each clock c,

$$\begin{aligned} z(c) = v(c)+1 \quad &\text{iff} \quad c \in X \text{ and } v(c) < k_{\max}, \\ v(c) < z(c) < v(c)+1 \quad &\text{iff} \quad c \notin X \text{ and } v(c) < k_{\max}, \\ z(c) > k_{\max} \quad &\text{iff} \quad v(c) = k_{\max}. \end{aligned}$$

Recall that $[\sigma]$ is also a subset of $(\mathbb{R}_+)^{\mathscr{C}}$. If $[v,X] \cap [\sigma] \neq \emptyset$, the transition (6) is added to $\mathscr{B}$. The action ϕ_v^X of $\mathscr{B}$ is derived from ϕ by replacing each pair $(p,Y) \in Q \times \mathscr{P}(\mathscr{C})$ appearing in ϕ with $((p,v_Y),Y)$, where v_Y is obtained from v^X by setting $v_Y(c) = 0$ for all $c \in Y$. Note that the structure of logical connectives in ϕ_v^X is the same as in ϕ.

A careful examination of the above construction reveals that the initial configuration of the automaton $\mathscr{B}$ should be treated differently, as no modification of v should be done in this case. We omit the details.

The automaton $\mathscr{B}$ is order-blind as required. It is deterministic (resp. nondeterministic, alternating) whenever the automaton $\mathscr{A}$ is so. The size of $\mathscr{B}$ may be exponential with respect to the size of $\mathscr{A}$, as the number of different sets X considered in (4) and (6) is exponential. □

Example 4.2. As an illustration, consider the nondeterministic one clock timed automaton that checks that there are two time stamps whose difference is 1 depicted in Figure 1. Nondeterminism is represented by separate arrows in the automaton instead of disjunctive formulae. For the sake of clarity, we omit some (non-accepting) transitions that would have to be added in order to fulfill the (Partition) condition. The

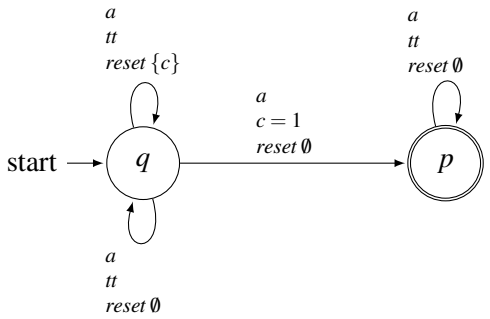


Figure 1: An automaton checking that there are two timestamp whose difference is 1.

construction described in the proof of Theorem 4.1 yields the order-blind register automaton of Figure 2.

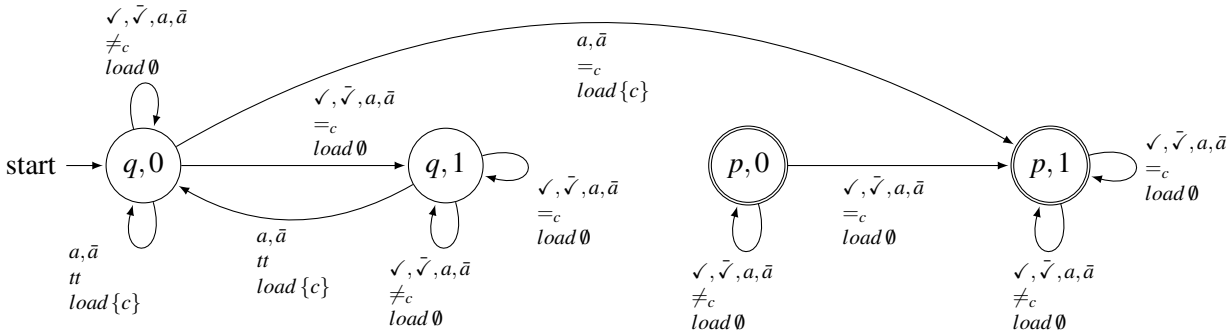


Figure 2: The automaton resulting from the construction of Theorem 4.1.

For the next results, we make use of the following Lemma.

Lemma 4.3. *The complement of the language of data braids is recognized by a nondeterministic one register automaton.*

Proof. A data word $w = (a_1, d_1) \cdots (a_n, d_n)$ fails to be a data braid iff either

- there is some marked (i.e., carrying an alphabet letter from $\bar{\mathbb{A}} \cup \bar{\checkmark}$) position $i+1$ such that $d_i \prec d_{i+1}$,
- there is some unmarked position $i+1$ such that $d_i \succeq d_{i+1}$,
- some datum strictly smaller than d_1 appears in w, or
- for some position i, there are two marked positions $j < k$, both greater than i, such that d_i does not appear among $\{d_j \dots d_k\}$; or if d_i does not appear after the last marked position.

A nondeterministic automaton can easily guess which of these conditions fails and verify it using one register. □

As a consequence of Lemma 4.3, the language of data braids is recognized by an alternating one register automaton. This is due to the fact that this model is closed under complementation.

We want to use Theorem 4.1 together with Lemma 4.3 to show Theorem 4.4 below. However, there is a subtle point here: by Lemma 4.3 register automata can recognize the complement of data braids, while we would need register automata to recognize the complement of the *image* of db(_) (a different language, since db(_) is not surjective). Unfortunately, the model cannot recognize such a language. In the proof below we deal with this problem by observing that db(_) is *essentially* surjective onto data braids.

Theorem 4.4. *The following decision problems for timed automata: language inclusion, language equality, nonemptiness and universality, reduce to the analogous problems for register automata. The reductions keep the number of registers equal to the number of clocks, and preserve the mode of computation (nondeterministic, alternating) of the input automaton.*

Proof. Consider the inclusion problem only, the other reductions are obtained in the same way. Given two timed automata $\mathscr{A}$ and $\mathscr{B}$, nondeterministic or alternating, we apply Theorem 4.1 to obtain two corresponding register automata $\mathscr{A}'$ and $\mathscr{B}'$. We claim that, for $\mathscr{A}_{\neg\mathrm{db}}$ given by Lemma 4.3, it holds:

$$\mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{B}) \quad \text{if and only if} \quad \mathscr{L}(\mathscr{A}') \subseteq \mathscr{L}(\mathscr{B}') \cup \mathscr{L}(\mathscr{A}_{\neg\mathrm{db}}).$$

(if) This implication is easy. Assume $\mathscr{L}(\mathscr{A}') \subseteq \mathscr{L}(\mathscr{B}') \cup \mathscr{L}(\mathscr{A}_{\neg\mathrm{db}})$ and let $w \in \mathscr{L}(\mathscr{A})$. By Theorem 4.1 $\mathrm{db}(w) \in \mathscr{L}(\mathscr{A}')$ and hence $\mathrm{db}(w) \in \mathscr{L}(\mathscr{B}')$. Again by Theorem 4.1 $w \in \mathscr{L}(\mathscr{B})$ as required.

(only if) Assume $\mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{B})$ and let $w \in \mathscr{L}(\mathscr{A}')$. If w is not a data braid then $w \in \mathscr{L}(\mathscr{B}') \cup \mathscr{L}(\mathscr{A}_{\neg\mathrm{db}})$ as required. Otherwise, w is a data braid, and we have the following:

Claim 4.4.1. There is a timed word v such that the automata $\mathscr{A}'$ and $\mathscr{B}'$ cannot distinguish between w and $\mathrm{db}(v)$, i.e., accept either both or none of them.

With the Claim above, by Theorem 4.1 we immediately obtain $v \in \mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{B})$. Again by Theorem 4.1 we get $\mathrm{db}(v) \in \mathscr{L}(\mathscr{B}')$, thus $w \in \mathscr{L}(\mathscr{B}')$ as well due to the Claim.

Proof of Claim 4.4.1. If the mapping db(_) was surjective onto data braids (up to isomorphism), then w would be equal to $\mathrm{db}(v)$ (up to isomorphism) for some v. This is however not the case! For example, consider appending $(\checkmark, d)$ for a sufficiently big d at the end of $\mathrm{db}(v)$. We then obtain a data braid which is not equal to $\mathrm{db}(v)$ for any v. But notice that in fact this last position is useless for $\mathscr{A}'$ and $\mathscr{B}'$.

A position i in a data braid w is considered *useless* iff (i) it is labeled by $(\checkmark, d)$, for some datum d, and all appearances of the datum d before i are labeled with $\checkmark$; or (ii) all the positions in its factor and in all following factors are labeled exclusively with $\checkmark$ or $\bar{\checkmark}$. Let $\widetilde{w}$ denote the result of removing all useless positions from w. We then have the following.

$$\widetilde{w} \text{ equals to } \mathrm{db}(v), \text{ up to isomorphism, for some timed word } v.$$

Consider any order preserving injection f from data values appearing in $\widetilde{w}$ to $[0,1)$, and let v be the result of the following steps: (1) i.e., replace every (a_i, d_i) of $\widetilde{w}$ with $(a_i, k + f(d_i))$, where k is the number of factors in $\widetilde{w}$ that end before position i; (2) remove all $\checkmark/\bar{\checkmark}$ positions; and (3) project the alphabet into $\mathbb{A}$.

By definition of db( ), $\mathrm{db}(v)$ is, up to isomorphism, equal to $\widetilde{w}$. Thus, $\mathscr{A}'$ and $\mathscr{B}'$ do not distinguish between $\widetilde{w}$ and $\mathrm{db}(v)$. It only remains to show that $\mathscr{A}'$ and $\mathscr{B}'$ do not distinguish between w and $\widetilde{w}$.

Each of $\mathscr{A}'$, $\mathscr{B}'$ either accepts both w and $\widetilde{w}$, or none of them.

This is true by construction, since when the input letter is $\checkmark$, the register automaton $\mathscr{A}'$ (or $\mathscr{B}'$) only updates the information about the integer part of those clocks c for which the equality test $=_c$ holds. When reading a useless position, if it is useless because of (i), then the equality holds for no clock; whereas in case (ii) the update will be inessential for the acceptance. This completes the proof of the claim as well as the proof of the theorem. □

5 From register automata to timed automata

In this section we complete the relation between the models of automata. We show that, up to a suitable encoding, languages of register automata may be recognized by timed automata. Again, this transformation keeps the number of registers equal to the number of clocks, and preserves the mode of computation (nondeterministic, alternating). Thus we obtain a tight relationship between the two classes of automata.

Theorem 5.1. *Given an alternating register automaton $\mathscr{A}$ one can compute in exponential time a timed automaton $\mathscr{B}$ such that for any data word w, $\mathscr{A}$ accepts w if and only if $\mathscr{B}$ accepts $tb(w)$. The number of clocks of $\mathscr{B}$ equals the number of registers of $\mathscr{A}$. Moreover, $\mathscr{B}$ is deterministic (resp. nondeterministic, alternating) if $\mathscr{A}$ is so.*

Proof. We describe the construction of a timed automaton $\mathscr{B}$ that faithfully simulates the behavior of a given register automaton $\mathscr{A}$. Let $\mathscr{R}$ be the set of registers of $\mathscr{A}$. The number clocks in $\mathscr{B}$ is the same as the number of registers in $\mathscr{A}$, $\mathscr{C} = \mathscr{R}$. A clock r is reset whenever $\mathscr{A}$ loads the current data value into register r. Moreover, each clock is also reset whenever the constraint $r = 1$ is met. Thus, when $\mathscr{B}$ runs over a time braid, no clock will ever have value greater than 1.

The state space of $\mathscr{B}$ is built on top of the states Q of $\mathscr{A}$. Additionally, for each clock r the automaton $\mathscr{B}$ stores in its state one bit of information describing whether the last marked position was seen before or after the last reset of r. This will allow $\mathscr{B}$ to simulate tests comparing the current data value with data values stored in registers.

Formally, states of $\mathscr{B}$ are pairs $(q,X) \in Q \times \mathscr{P}(\mathscr{R})$. Initially the set X, which we call the register component of a state, is chosen as $X = \mathscr{R}$ (according to the assumption that the register automaton loads the first datum into all its registers as its very first action). At each marked symbol $\bar{a}$ or $\bar{\checkmark}$, the automaton $\mathscr{B}$ sets $X := \emptyset$. Moreover, at each reset of r (at marked or unmarked positions), r is added to X. As a consequence of this behavior, it invariantly holds: $r \in X$ if and only if the position of the last reset of r is greater or equal to the last marked position. Hence, the test $\preceq r$ (current data smaller or equal to register r) is satisfied at a state (q,X) if and only if $r \notin X$. The table below summarizes all the atomic tests and the corresponding constraints on clock values and on the register component of state:

test in $\mathscr{A}$	meaning	constraint in $\mathscr{B}$
$\preceq r$	current datum smaller or equal to r	$r \notin X$
$\prec r$	current datum smaller than r	$r \notin X$ and $r \neq 1$
$\succeq r$	current datum greater or equal to r	$r \in X$ or $r = 1$
$\succ r$	current datum greater than r	$r \in X$

We just described how the register component of a state is updated and when the clocks are reset. At input letter $\checkmark$ or $\bar{\checkmark}$ this is the only the automaton $\mathscr{B}$ has to do. Each transition $q,a,t \mapsto b$ of $\mathscr{A}$ with $a \in \mathbb{A}$ gives raise to a number of transitions $(q,X),a,\sigma \mapsto b'$ and $(q,X),\bar{a},\sigma \mapsto b'$ of $\mathscr{B}$ that additionally

keep track of the change of state of $\mathscr{A}$ and of load operations performed by $\mathscr{A}$, as described above. The register test t gives raise to a clock constraint σ as described in the table above. The structure of logical connectives in b' is the same as in b, hence $\mathscr{B}$ is deterministic (resp. nondeterministic, alternating) whenever $\mathscr{A}$ is so. □

Example 5.2. Consider the simple nondeterministic one register automaton that checks if the first datum in a word is equal to the last one depicted in Figure 3. Similarly as before, the (Partition) condition is not satisfied as some (non-accepting) transitions are missing. The construction in the proof of Theorem 5.1

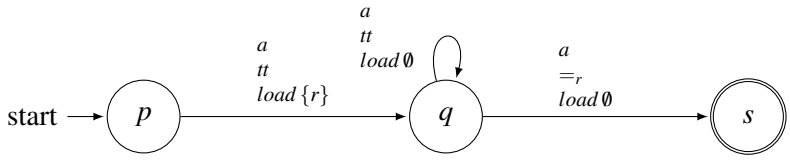


Figure 3: An automaton checking that the first datum is equal to the last one.

yields the automaton of Figure 4.

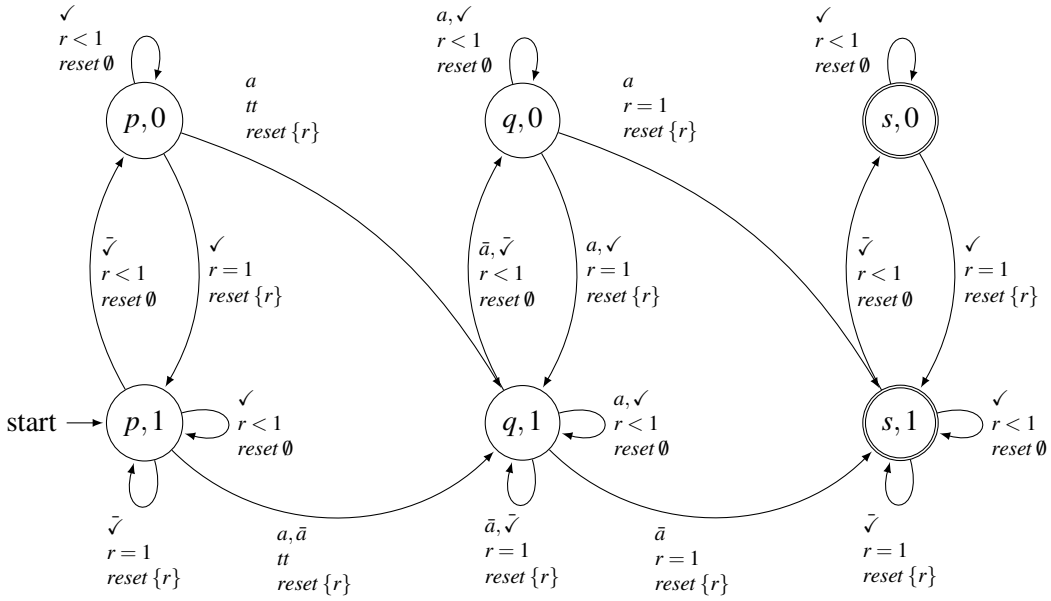


Figure 4: The result of the construction of Theorem 5.1.

For the next results, we make use of the following Lemma.

Lemma 5.3. *The complement of the language of timed braids is recognized by a nondeterministic one clock automaton.*

Proof. A timed word fails to be a timed braid iff: (0) the time-stamp in the first position is not equal 0; (1) there is some marked position appearing at some non-integer position; (2) there is some unmarked position appearing at some integer position; (3) there is some missing position with time-stamp t induced by the existence of an element $(a, t-1.0)$ for some a.

It is easy to check that (0), (1) and (2) can be verified by a deterministic one clock automaton. (3) is verified as follows. The automaton guesses a position i, where it resets the clock. Then it checks that after the next marked position, the clock is continuously strictly smaller than 1 until it finally becomes strictly greater than 1, or the word ends. □

As a consequence of Lemma 5.3, the language of timed braids is recognized by an alternating one clock automaton. This is due to the fact that this model is closed under complementation.

The following theorem is proved analogously to Theorem 4.4, using Theorem 5.1 and Lemma 5.3:

Theorem 5.4. *The following decision problems for register automata: language inclusion, language equality, nonemptiness and universality, reduce to the analogous problems for timed automata. The reductions keep the number of clocks equal the number of registers, and preserve the mode of computation (nondeterministic, alternating) of the input automata.*

Sketch. The proof is very similar to the proof of Theorem 4.4. We need to deal with a minor difficulty of the same kind: the mapping $\mathrm{tb}(\_)$ is not surjective, up to isomorphism, onto timed braids. Thus we need the following claim, similar to one used in the proof of Theorem 4.4:

Claim 5.4.1. For each timed braid w there is a data word v such that the automata $\mathscr{A}'$ and $\mathscr{B}'$ cannot distinguish between w and $\mathrm{tb}(v)$.

The claim is demonstrated similarly as before, by identifying *useless* positions in a timed braid. First, a position i is *useless* iff (1) it is labeled with $(\checkmark, t)$, for some (necessarily non-integer) t, and all positions before i carrying time-stamps with the same fractional part as t are labeled with $\checkmark$; or (2) position i carries time stamp t and all positions carrying time-stamps in the interval $[t, t+1)$ are labeled exclusively with $\checkmark$ or $\bar{\checkmark}$.

Similarly as before, the automaton obtained by Theorem 5.1 cannot tell the difference when all useless positions are removed from a word. To see this, observe that in case (1) no clock will be reset as the constraint $r = 1$ will not hold for any clock r. In case (2), the whole segment of length 1 labeled exclusively by $\checkmark$ and $\bar{\checkmark}$ may be safely skipped as it has no impact on state of the automaton. □

6 Applications

Here we provide some evidence that the tight relationship between register automata and timed automata may be useful: we transfer a couple of results from timed do data setting. First, from the fact that 1-clock alternating timed automata have decidable emptiness [9], applying Theorem 5.4 we obtain:

Theorem 6.1. *The emptiness problem for alternating one register automata is decidable.*

Note that register automata, as we define it here, work over *ordered* data domains and are capable of comparing data values w.r.t. $\preceq$. According to our terminology, decidability was only known for the subclass of order-blind automata [4]. Interestingly, the results holds for *any* total order over data.

A note on complexity Note that alternating 1 register automata over an ordered domain have the same complexity (modulo an exponential time reduction) as alternating 1 clock timed automata. However, we must remark that these automata have a much higher complexity than *order blind* alternating 1 register automata, although both are beyond the primitive recursive functions. While the latter can be roughly bounded by the Ackermann function applied to the number of states, the complexity of the former majorizes every multiply-recursive function (in particular, Ackermann's).

More precisely, the emptiness problem for alternating timed automata with 1 clock sits in the class $\mathfrak{F}_{\omega^\omega}$ in the Fast Growing Hierarchy [12]—an extension of the Grzegorczyk Hierarchy for non-primitive recursive functions—by a reduction to Lossy Channel Machines [1], which are known to be 'complete' for this class, i.e. in $\mathfrak{F}_{\omega^\omega} \setminus \mathfrak{F}_{<\omega^\omega}$ [3]. However, the emptiness problem for *order blind* alternating 1 register automata belongs to $\mathfrak{F}_\omega$ in the hierarchy, by a reduction to Incrementing Counter Automata [4], which are complete for $\mathfrak{F}_\omega$ [18, 5]. We then obtain the following result.

Corollary 6.2. *The emptiness problem of alternating 1 register automata over a linearly ordered domain is in $\mathfrak{F}_{\omega^\omega} \setminus \mathfrak{F}_{<\omega^\omega}$ in the Fast Growing Hierarchy.*

We show another example of a result that can be directly copied from the timed to the data setting:

Theorem 6.3. *Consider the following inclusion problem: Given a nondeterministic register automaton $\mathscr{A}$ (with possibly many registers) and an alternating one register automaton $\mathscr{B}$, is every word accepted by $\mathscr{A}$ also accepted by $\mathscr{B}$? This problem is decidable and of non-primitive recursive complexity.*

The result is immediate as the analogous problem is decidable and non-primitive recursive for timed automata [9]. Decidability follows from Theorem 5.4 while the lower bound follows from Theorem 4.4.

As the last example of application of our results, we consider the emptiness problem over a restricted class of data words. We say that a data word $(a_1,d_1)\dots(a_n,d_n)$ is *m-decreasing* iff there are at most $m-1$ positions i with $d_i \succeq d_{i+1}$.

Theorem 6.4. *Let m be a fixed nonnegative number. The non-emptiness problem for alternating register automata over m-decreasing data words is decidable and non-elementary.*

Again, the result is an immediate consequence of the result of [7]: emptiness of alternating timed automata over m-bounded timed words is decidable and non-elementary, where m-bounded words are those with all time stamps smaller than m. The upper bound follows by Theorem 5.1 and from the fact that a data word w is m-decreasing iff $\mathrm{tb}(w)$ is m-bounded. Conversely, the lower bound follows from Theorem 4.1 and from the symmetric fact: a timed word is m-bounded iff $\mathrm{db}(w)$ is m-decreasing.

Finally, note that the upper (decidability) bounds of Theorems 6.3 and 6.4 also apply to the class of order-blind register automata.

7 Discussion

We have shown that timed and register automata on finite words are essentially equivalent. In order to relate these two models we introduced the notion of a 'braid'-like structure, that corresponds naturally to the way a clock works when running over a timed word. As shown, most decision problems are actually equivalent for these two models. This work can be useful to derive results on one model of automata as corollaries of results on the other model, by exploiting the duality between timed and data automata shown here. This is evidenced here by showing some new results (Section 6). One limitation of our results is the both translations between timed and register automata suffer of an exponential blow-up in the size of the automaton. As a consequence, one should not expect applicability to low-complexity problems, like non-emptiness of timed or register automata, in both cases a PSPACE-complete problem.

As a relatively straightforward further step, we plan to investigate automata over ω-words; in particular, we hope for transferring results of [17, 16] to register automata and for comparing them e.g. with those of [11]. It seems however that one must restrict to dense orders on data domain for proper treatment of infinite words. Furthermore, we also plan to attempt a similar comparison of logical formalisms, namely of freeze LTL and MTL or TPTL, both over finite and ω-words. This should allow to compare or transfer the complexity result for syntactic fragments of freeze LTL and MTL that appeared recently in the literature, see e.g. [17, 16, 10, 6].

Acknowledgements The authors thank the anonymous reviewers for their valuable comments.

References

[1] Parosh Aziz Abdulla, Johann Deneux, Joël Ouaknine & James Worrell (2005): *Decidability and Complexity Results for Timed Automata via Channel Machines*. In: *ICALP*, pp. 1089–1101. Available at `http://dx.doi.org/10.1007/11523468_88`.

[2] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theoretical Computer Science* 126, pp. 183–235.

[3] Pierre Chambart & Philippe Schnoebelen (2008): *The Ordinal Recursive Complexity of Lossy Channel Systems*. In: *LICS*, IEEE Computer Society Press, pp. 205–216. Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CS-lics08.pdf`.

[4] Stéphane Demri & Ranko Lazić (2009): *LTL with the freeze quantifier and register automata*. *ACM Trans. Comput. Log.* 10(3). Available at `http://doi.acm.org/10.1145/1507244.1507246`.

[5] Diego Figueira, Santiago Figueira, Sylvain Schmitz & Philippe Schnoebelen (2010): *Ackermann and Primitive-Recursive Bounds with Dickson's Lemma*. *ArXiv e-prints* Available at `http://arxiv.org/abs/1007.2989`.

[6] Diego Figueira & Luc Segoufin (2009): *Future-Looking Logics on Data Words and Trees*. In: *MFCS*, pp. 331–343. Available at `http://dx.doi.org/10.1007/978-3-642-03816-7_29`.

[7] Mark Jenkins, Joël Ouaknine, Alexander Rabinovich & James Worrell (2010): *Alternating Timed Automata over Bounded Time*. In: *LICS*, IEEE Computer Society Press.

[8] Michael Kaminski & Nissim Francez (1994): *Finite-Memory Automata*. *Theor. Comput. Sci.* 134(2), pp. 329–363.

[9] Sławomir Lasota & Igor Walukiewicz (2008): *Alternating timed automata*. *ACM Trans. Comput. Log.* 9(2).

[10] Ranko Lazić (2006): *Safely Freezing LTL*. In: *FSTTCS*, pp. 381–392. Available at `http://dx.doi.org/10.1007/11944836_35`.

[11] Ranko Lazić (2008): *Safety alternating automata on data words*. *CoRR* abs/0802.4237. Available at `http://arxiv.org/abs/0802.4237`.

[12] M.H. Löb & S.S. Wainer (1970): *Hierarchies of number theoretic functions, I*. *Archiv für Mathematische Logik und Grundlagenforschung* 13, pp. 39–51.

[13] Frank Neven, Thomas Schwentick & Victor Vianu (2004): *Finite state machines for strings over infinite alphabets*. *ACM Trans. Comput. Log.* 5(3), pp. 403–435. Available at `http://doi.acm.org/10.1145/1013560.1013562`.

[14] Joël Ouaknine & James Worrell (2004): *On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap*. In: *LICS*, IEEE Computer Society Press, pp. 54–63. Available at `http://csdl.computer.org/comp/proceedings/lics/2004/2192/00/21920054abs.htm`.

[15] Joël Ouaknine & James Worrell (2005): *On the Decidability of Metric Temporal Logic*. In: *LICS*, IEEE Computer Society Press, pp. 188–197. Available at `http://dx.doi.org/10.1109/LICS.2005.33`.

[16] Joël Ouaknine & James Worrell (2006): *Safety Metric Temporal Logic Is Fully Decidable*. In: *TACAS*, pp. 411–425. Available at `http://dx.doi.org/10.1007/11691372_27`.

[17] Paweł Parys & Igor Walukiewicz (2009): *Weak Alternating Timed Automata*. In: *ICALP*, pp. 273–284. Available at `http://dx.doi.org/10.1007/978-3-642-02930-1_23`.

[18] Philippe Schnoebelen (2010): *Revisiting Ackermann-hardness for lossy counter systems and reset Petri nets*. In: *MFCS 2010*, *LNCS* 6281, Springer. Available at `http://www.lsv.ens-cachan.fr/~phs`.