Technical Report No. 99-428

Towards a Meaningful Formal Definition of Real–Time Computations*

Stefan D. Bruda and Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario, K7L 3N6 Canada
Email: {bruda,akl}@cs.queensu.ca

July 8, 1999

Abstract

There are few formal definitions of real–time problems, and the currently available definitions do not capture all the relevant aspects of such computations. We propose a new definition, and we believe that it allows a unified treatment of all practically meaningful variants of real–time computations. In order to support our thesis, we also present some important features of real–time algorithms, namely the presence of deadlines and the continuous arrival of input data, together with their corresponding models in our formalism. We believe that this is a first step towards a unified and realistic complexity theory for real–time computation.

Key words and phrases: Real–time computation, complexity theory, formal languages, timed languages, ω-languages, parallel complexity theory.

1 Introduction

The area of real–time computations has a strong practical grounding, in domains like operating systems, databases, and the control of physical processes. Besides these practical applications, however, research in this area is primarily focused on formal methods and on communication issues in distributed real–time systems.

*This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Little work has been done in the direction of algorithms and complexity theory. In fact, the limited extent of this work is emphasized by the fact that even a realistic general definition for real–time algorithms is missing, although implicit definitions can be found in many places. Some papers have tried to address this issue, providing abstract machines that model real–time algorithms. In this context, one can notice the real–time Turing machine, proposed for the first time in [25] and further studied in [13, 21, 23]. Such a formalism offers many insights into the theory of real–time systems, but it fails to capture many other aspects that are important in practice. Another model is the real–time producer/consumer paradigm, proposed in [15], which takes into account some important features, but is suitable for modeling certain real–time systems rather than for developing a general complexity theory. Finally, the concept of timed automata is introduced in [8]. The format of languages accepted by such devices is also presented, together with their closure properties. However, the power of the language families analyzed in [8] is limited, since there are real–time problems that cannot be formalized as languages recognizable by memoryless finite state models.

Indeed, the domain of real–time systems is very complex, with requirements varying from application to application. For example, while in some applications the real–time component is the presence of deadlines imposed upon the computation, other applications require that input data are processed as soon as they become available, with more data to come while the computation is in progress. Variants (and combinations) of these two main requirements are often present. This complexity of the domain is probably the main obstacle towards a unified theory.

In this paper, we try to address this issue. We believe that the model of timed languages proposed in [8] is a powerful tool, but the device used as acceptor (namely, a finite automaton) is rather weak. We suggest therefore an extension of this study. More precisely, we keep most of the important ingredients in the definition of timed languages from [8], but we apply such a definition to a larger extent, suggesting a general model for the acceptors of such languages. We believe that our construction captures all the practical aspects of real–time computations, and we support our thesis by showing how two of the main ingredients of such computations (namely, computing with deadlines, and input data that are not available entirely at the beginning of computation) can be modeled using our formalism. We also believe that, starting from the definitions outlined in this paper, a unified complexity theory for real–time systems can be naturally developed.

We organize the paper as follows. In the next section we briefly summarize the notations used through the paper. Then, in section 3, we summarize the existing definitions, emphasizing the points where they fail to capture all the relevant practical aspects. Then, in section 4 we introduce a new definition, which is more general and more flexible. We also present here some important features of real–time algorithms, together with their models in our formalism. We conclude in the last section.

2 Notations

Given some finite alphabet Σ, the set of all the words of finite (but not necessary bounded) length over Σ is denoted by Σ^*. The cardinality of $\mathbb{N}$, the set of natural numbers, is denoted by ω. Then, the set Σ^ω contains exactly all the words over Σ of length ω. Given two words σ_1 and σ_2, $\sigma_1\sigma_2$ denotes the concatenation of them. The length of a word σ is denoted by $|\sigma|$. $\mathbb{R}$ denotes the set of real numbers.

A general finite automaton is a tuple $A = (\Sigma, S, s_0, \delta, F)$, where Σ is the (finite) input alphabet, S is a (finite) set of states, s_0 is the initial state, δ is the transition relation, $\delta \in S \times S \times \Sigma$, and F is the set of accepting states, $F \subseteq S$. When we use Σ, S, s_0, δ, and F, we imply the above meaning of these symbols unless otherwise specified. The accepting condition for an usual finite automaton A is as follows: If at the end of the input string, A is in some state from F, then the input is accepted. Otherwise, the input is rejected.

We assume that the reader is familiar with the concept of Turing machines [18]. The transition function of the configurations of a Turing machine M is denoted by $\vdash_M$, with the subscript possibly omitted when there is no ambiguity. As usual, $\vdash_M^*$ denotes the reflexive and transitive closure of $\vdash_M$.

3 Previous Work

3.1 Real–Time Turing Machines

Probably the first work on formalizing the notion of real–time is [25]. Here, the notion of *real–time Turing machine* is introduced. Then, the family of functions/languages that are computed/recognized by such machines is inferred. This direction is further pursued in [21, 23].

Definition 3.1 [23]

1. For some constant k, $k \geq 1$, an *on–line Turing machine* is a deterministic k-tape Turing machine M whose set of states is divided into two subsets: the set of *polling* states K_p and the set of *autonomous* states K_a. All the states that lead to h in one step are polling states, and the initial state is a polling state. In addition, the head is allowed to move only on the right on the input tape, and the relation $\vdash_M$ has the following property: if $q \in K_p$, $q'' \in K_a$, and $q' \in K_p \cup K_a$, then

$$\begin{aligned}
(q, u\underline{a}bv, x_1, \ldots, x_k) &\vdash_M (q', ua\underline{b}v, x_1', \ldots, x_k'),\\
(q'', u\underline{a}bv, x_1, \ldots, x_k) &\vdash_M (q', u\underline{a}bv, x_1', \ldots, x_k'),\\
(q, u\underline{a}bv, x_1, \ldots, x_k) &\vdash_M (h, u\underline{a}bv, x_1', \ldots, x_k').
\end{aligned}$$

   M accepts the input w iff $(s, w, x_1, \ldots, x_k) \vdash_M^\tau (h, \lambda, x_1, \ldots, x_k)$, where s is the initial state and $\tau > 0$ is called the *running time* of M on w.

2. A *real–time Turing machine* is an on–line Turing machine for which $K_a = \emptyset$. A language accepted by such a machine is called a *real–time definable language*.

□

Briefly, a real–time Turing machine is an on–line deterministic Turing machine, whose running time is n, the length of the input word. Note that, conforming to a result from [13], given a deterministic on–line Turing machine that recognizes some language L and whose running time is $O(n)$, one can construct a real–time Turing machine that recognizes L. A language recognized by a real–time Turing machine is called *real–time definable*.

Almost the same definition, this time in terms of algorithms rather than Turing machines, can be found in [22]. Here, a *linear–time algorithm* runs in $O(n)$ steps on any input of length n. A real–time algorithm is a linear–time algorithm which has the additional requirement that it spends only $O(1)$ steps on any input symbol.

Note that the main difference between this definition and the definition in terms of Turing machines is the absence of the on–line requirement. Therefore, a real–time algorithm conforming to this definition may have access to all the input data at the beginning of the computation. Moreover, such an algorithm may skip some input data. However, such a definition seems not to be supported by practice. Indeed, in most of the real–time applications, such as real–time databases [7, 24], real–time scheduling [14], tracking devices [15], or process control [17], the input data cannot be skipped. As well, not all of them are available at the beginning of the computation.

In addition, in the real world, $O(1)$ time for each input datum is not always a sufficiently strong condition. As an example, take the railroad crossing problem [17], which consists in the design of a controller that opens and closes a gate at a railway crossing. The specifications of the problem impose precise time limits on the actions performed by the controller. For example, it is not mentioned that the gate should close at some constant (but arbitrarily large) time after the request to close has been issued, but the action has to be completed in some fixed time span (say, 20 seconds) instead.

3.2 The Real–Time Producer/Consumer Paradigm

Another model for real–time computations is presented in [15]. This model is based on the producer/consumer paradigm. In such a paradigm, there are two entities, a producer, that produces messages, and a consumer, that consumes the produced messages. They communicate through a buffer, that keeps those messages that were produced, but not consumed yet. Based on this model, the *real–time producer/consumer paradigm* (RTP/C) is introduced. Here, the producer produces messages at a given rate, and the consumer must consume the messages at the rate they are produced (the buffer is thus eliminated). A real–time system is composed then by a set of such communicating processes, together with some storage space.

The thesis mentioned in [15] is that the RTP/C paradigm applies to a wide variety of interesting and important real–time applications, where all the data arriving from the external world must be processed in real–time. However, the concept of production rate may not be expressive enough in some cases. More precisely, given the railway crossing problem mentioned above, the main event is the arrival of a train at the crossing, which does not happen at a specified rate (in fact, there is a possibility that the train never arrives). Another example where the RTP/C paradigm is not applicable is the data–accumulating paradigm (described in section 4.3.2), where the arrival rate varies over time.

3.3 Timed Automata

Finally, a third model of real–time computation is the *timed (finite) automaton* [8]. The theory of such automata starts from the theory of ω-automata.

An *ω-automaton* is a usual finite state automaton $A = (\Sigma, S, s_0, \delta, F)$, whose accepting condition is modified, in order to accommodate input words of infinite length. More precisely, given an (infinite) word $\sigma = \sigma_1\sigma_2, \ldots$, the sequence

$$r = s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} \ldots$$

is called a *run* of A over σ, provided that $(s_{i-1}, s_i, \sigma_i) \in \delta$ for all $i > 0$. For such a run, $\inf(r)$ is the set of all the states s such that $s = s_i$ for infinitely many i.

Regarding the accepting condition, a *Büchi automaton* has a set $F \subseteq S$ of accepting states. A run r over a word $\sigma \in \Sigma^\omega$ is accepting iff $\inf(r) \cap F \neq \emptyset$. The acceptance of a *Muller automaton* on the other hand does not use the concept of final state. For such an automaton, an *acceptance family* $\mathcal{F} \subseteq 2^S$ is defined. Then, a run r over a word σ is an accepting run iff $\inf(r) \in \mathcal{F}$. A language accepted by some automaton (Büchi of Muller) consists of the words σ such that the automaton has an accepting run over σ.

Another ingredient of the theory developed in [8] is the *time sequence*. A time sequence $\tau = \tau_1\tau_2\ldots$ is an infinite sequence of positive real values, such that the following constraints are satisfied: (i) *monotonicity*: $\tau_i \leq \tau_{i+1}$ for all $i \geq 0$, and (ii) *progress*: for every $t \in \mathbb{R}$, there is some $i \geq 1$ such that $\tau_i > t$. Then, a *timed ω-word* over some alphabet Σ is a pair (σ, τ), where $\sigma \in \Sigma^\omega$, and τ is a time sequence. That is, a timed ω-word is an infinite sequence of symbols, where each symbol has a time value associated with it. The time value associated to some symbol can be considered the time at which the corresponding symbol becomes available. A timed ω-language is a set of timed ω-words.

A *clock* is a variable over $\mathbb{R}$, whose value may be considered as being externally modified. Given some clock x, two operations are allowed: reading the value stored in x, and resetting x to zero. At any time, the value stored in x corresponds to the time elapsed from the moment that x has been most recently reset. For a set X of clocks, a set of *constraints* over X, $\Phi(X)$, is defined by: d is an element of $\Phi(X)$ iff d has one of the following forms: $x \leq c$, $c \leq x$, $\neg d_1$, or $d_1 \wedge d_2$, where c is some constant, $x \in X$, and $d_1, d_2 \in \Phi(X)$.

Starting from these notions, the notion of timed ω-regular languages is introduced. A *timed Büchi automaton* (TBA) is a tuple $A = (\Sigma, S, s_0, \delta, C, F)$,

where C is a finite set of clocks. This time, the transition relation δ is defined as $\delta \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$. An element of δ has the form (s, s', a, l, d), where l is the set of clocks to be reset during the transition, and d is a clock constraint over C. The transition is enabled only if d is valued to true using the current values of the clocks in C.

A run r of a TBA $A = (\Sigma, S, s_0, \delta, C, F)$ over some timed ω-word (σ, τ) is an infinite sequence of the form

$$r = (s_0, \nu_0) \stackrel{\sigma_1,\tau_1}{\longrightarrow} (s_1, \nu_1) \stackrel{\sigma_2,\tau_2}{\longrightarrow} (s_2, \nu_2) \stackrel{\sigma_3,\tau_3}{\longrightarrow} \cdots, \tag{1}$$

where $\sigma = \sigma_1\sigma_2\ldots$, $\tau = \tau_1\tau_2\ldots$, $\nu_i \in \{f | f : C \to \mathbb{R}\}$ for all $i \geq 0$, and the following conditions hold:

- $\nu_0(x) = 0$ for all $x \in C$,
- for all $i \geq 0$, there is a transition $(s_{i-1}, s_i, \sigma_i, i_i, d_i) \in \delta$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies d_i, for all $x \in C - l_i$, $\nu_i(x) = \nu_{i-1}(x) + \tau_i - \tau_{i-1}$, and, for all $x' \in l_i$, $\nu_i(x') = 0$.

The notions of accepting run, and language accepted by a TBA are defined similarly to the case of Büchi automata.

A timed ω-language accepted by some TBA will be called a *timed ω-regular language*. Note that the name for such languages in [8] is simply *timed regular languages* (as well, a timed ω-language is denoted by timed language), but we prefer this terminology for reasons that will become evident in the next section, where we use both notions of finite and infinite timed words.

However, the TBA used in [8] for recognition of timed (ω-)languages is not sufficiently powerful to take into account all the real–time applications. But we will postpone this discussion till the next section.

4 Timed Languages

While the notion of timed languages is very powerful, the device used for recognition of such languages in [8] (that is, a finite–state timed automaton) is not powerful enough to model all the real–time computations that are meaningful in practice. This is supported by the following immediate result.

Theorem 4.1 *There are languages formed by infinite words (ω-languages) that are not ω-regular.*

Proof. Let us consider the following language over the alphabet $\Sigma = \{a, b, c, d\}$: $L = \{a^u b^x c^v d^x | u, x, v > 0\}$. It is immediate that L is not regular. Now, consider the following ω-language: $L_\omega = \{l_1\$l_2\$l_3\$\ldots | l_i \in L \text{ for any } i > 0, \text{ and } \$ \notin \Sigma\}$.

Assume now that L_ω is ω-regular. Then, there is a Büchi automaton $A = (\Sigma, S, s_0, \delta, F)$ that recognizes it. Let x be a word in L_ω, $x = x_1\$x_2\$x_3\$\ldots$. Therefore, there is a run r of A over x such that $\inf(r) \cap F \neq \emptyset$.

In the run r, let S_1 be the set of all the states that A is into immediately after parsing a symbol \$, and S_2 the set of all states A is into immediately before parsing a symbol \$. But then one can construct a finite automaton A' that recognizes L: let the initial state of A' be some $s' \notin S$; then, the set of states of A' is $S \cup \{s'\}$, the set of final states of A' is S_2, and the transition function of A' is δ, augmented with λ-transitions from s' to each state in S_1.

But this is clearly a contradiction, since L is not regular. □

Corollary 4.2 *There are timed ω-languages that are not (timed) ω-regular.*

Proof. Simply attach to each word in the language L_ω a time sequence $\tau = \tau_1\tau_2\ldots$, such that $\tau_1 = 0$, and $\tau_{i+1} - \tau_i = 1$ for any $i \geq 1$. □

Note that the language L_ω built in the proof of theorem 4.1 is not uninteresting from a practical point of view. Indeed, it models a search into a database for a given key: the database is modeled by the word $a^u b^x c^v$, the key to search for is d^x, and the instance that matches the query is simulated by b^x. We just found hence some practical situation which does not pertain to the class of (timed) ω-regular languages.

4.1 A Formal Definition

Despite the limited scope of the finite state approach, the concept of timed languages is a very powerful one. We propose therefore a definition that is similar to the one in [8], but is not restricted to finite state acceptors.

Definition 4.1 1. A (finite) *timed word* over some alphabet Σ is a pair $(\sigma\#, \tau')$, where $\sigma \in \Sigma^*$, $\# \notin \Sigma$, τ is a time sequence, $\tau' \subseteq \tau$, and $|\sigma| + 1 = |\tau|$. A timed language over Σ is a set of timed words over Σ.

2. A *timed ω-word* over Σ is a pair (σ, τ), $\sigma \in \Sigma^\omega$, and τ is a time sequence. A timed ω-language over Σ is a set of timed ω-words over Σ.

□

Definition 4.1 is the same as the definition in [8], except that we consider finite timed languages as well. However, while the study in [8] restricted itself to those timed ω-languages that are recognized by finite state acceptors, our suggestion is that other acceptors (with unbounded storage space) should be considered. We offered a motivation of this by corollary 4.2. In the case of finite timed words, a termination time on the recognition process should be imposed. This is achieved by the presence of a special symbol # appended at the end of the word, with an associate time stamp $\tau_\#$, which is the deadline for the recognition process.

In light of the above definition, we can also establish the general form of an acceptor for timed languages. Extending the idea from [8], we define a (general) acceptor $\mathcal{A}$ for timed languages as being composed of a finite control, an input device, and a finite set of clocks, as defined in the previous section. The acceptor

may have access to an infinite amount of memory. However, only a finite amount of this memory should be used in each computation. The finite state control has a designated "final" state f. In the case of ω-languages, a run of $\mathcal{A}$ over some word σ is defined analogously to the run of a TBA (see equation (1)), except that the clocks are not updated only at the arrival of a new input symbol, but at each execution of an elementary operation instead. More precisely, a run of an acceptor is a sequence of the form

$$r = s_0 \stackrel{\sigma_1,\tau_1}{\longrightarrow} s_1 \stackrel{\sigma_2,\tau_2}{\longrightarrow} s_2 \stackrel{\sigma_3,\tau_3}{\longrightarrow} \cdots, \tag{2}$$

where $\sigma = \sigma_1\sigma_2\ldots$, $\tau = \tau_1\tau_2\ldots$, and s_0, s_1, ... are states of the acceptor (s_0 being the initial state). The clocks are defined as in section 3.3, in the sense that the only two operations allowed for some clock x is reading of the value stored in x and resetting x to zero. However, we claimed that the clocks may be considered externally modified. In the case of TBAs, this condition means that each time a new symbol appears at the input, the difference between the timestamp of that symbol and the timestamp of the symbol that preceded it is added to all the clocks, as expressed in the definition of a run of a TBA (see equation (1)). Indeed, since the transitions of a finite automata can be considered as taking a time unit to execute, it is enough to update the clocks at the arrival of a new symbol only. In other words, a TBA can consider every input at the precise time it arrives. However, when more complex acceptors are considered, the internal processes for an input symbol may last longer than the time between the arrival of that symbol and the arrival of its successor. Therefore, we consider that each clock is incremented each time an elementary operation is executed. A clock may be reset only at the time some input symbol is read though.

Then, $\mathcal{A}$ accepts the timed language L if, for any input timed word $(\sigma\#, \tau')$, there is a computation of $\mathcal{A}$ that reaches the state f at time $\tau_\#$ iff $(\sigma\#, \tau') \in L$. Analogously, an acceptor $\mathcal{A}$ accepts a timed ω-language L' if, for any timed ω-word (σ, τ), there is a run r of $\mathcal{A}$ over (σ, τ) such that $f \in \inf(r)$ iff $(\sigma, \tau) \in L'$. In what follows we shall call an acceptor for a timed language *timed acceptor*, and an acceptor for a timed ω-language a *ω-timed acceptor*.

Even if we discussed here only the notion of timed languages, the extension for timed problems is immediate. Indeed, a timed problem can be defined as a problem whose possible inputs form a timed language. The definition for a timed ω-problem is similar. Concerning the form of a machine that solves an timed (ω-)problem, it is an acceptor for the corresponding timed (ω-)language, except that it is equipped with an output device, where the solution of the problem eventually becomes available. However, we will allow the machine to write to the output device only if it's finite control is in the "final" state f.

A final note on the set of clocks is in order. In the general case, since the acceptor has access to an unlimited storage space, the clocks can be stored here, and no reference to them is necessary. However, the storage space may be limited. For example, we presented in section 3.3 a special case of such acceptors, that fall in our general characterization, except that the storage capacity is

null. Similarly, one can define timed push–down automata, where the storage capacity, even if unbounded, has a stack structure, which is not suitable for storing an arbitrary number of clocks. Therefore, we preferred to treat the clocks in a special manner, and not make them part of the main memory.

4.2 Timed Acceptors and the On–Line Property

There are few formal definitions for on–line algorithms, although this notion is widely used. We already presented the definition from [23], which is given terms of Turing machines, but it can be easily extended to other models (definition 3.1, item 1).

In other words, an on–line algorithm must process all the input data in the order they come, without any information on the future data. We will use this definition, except that we drop the requirement conforming to which the algorithm is deterministic. However, even in this weaker form, the definition is still too restrictive to be useful in our theory of timed languages. Take for example the language L_ω from the proof of theorem 4.1. It is clear that an algorithm that accepts this language is not on–line. Indeed, let the currently considered part of the input word be $\ldots \$a^u b^x c^v$. It is clear then that no decision about the acceptance or rejection of the current string can be made before x d's have been read.

However, it is easy to see that any acceptor of a timed ω-language processes the input in bundles. More precisely, a bundle is delimited by the moments when the acceptor reaches the "final" state f. Moreover, because of the definition of the accepting run, the number of such bundles is infinite. Such an algorithm is not necessarily on–line, but the features are similar, in the sense that the algorithm is limited in its knowledge about future data to the current bundle instead of the current datum. We will call such a property *pseudo–on–line*. That is, we showed that the definition of real–time definable languages from [25] is too restrictive, because of the on–line requirement. We suggest that this condition should be replaced by the pseudo–on–line one.

4.3 Examples

Our thesis is that the theory of timed languages covers all the practically meaningful aspects of real–time computations, while doing so in a formal, unified manner.

In particular, note that all the formal models summarized in section 3 can be considered particular cases of this more general form. More precisely, a real–time definable language is a timed language, where, for an input of length n, the time values are $(\tau_1, \tau_2, \ldots, \tau_n)$, such that $\tau_i - \tau_{i+1}$ is constant for any i. Next, the RTP/C paradigm can be modeled by creating the time sequence according to the rate at which the messages are emitted (however, while the RTP/C model is most suitable for program specification and verification (as mentioned in [15]), the model of timed languages is more adequate for complexity theoretic

approaches). Finally, timed automata are obviously a particular case of timed ω-acceptors.

In order to further support our thesis, we will take some meaningful examples, with practical applications, and we will construct timed ω-languages that model them.

In general, given some problem, we denote the input and the output alphabets by Σ and Δ, respectively. We also denote by n and m the sizes of the input ι and of the output o. When a timed ω-word is denoted by (σ, τ), we consider that $\sigma = \sigma_1\sigma_2\ldots$, and $\tau = \tau_1\tau_2\ldots$. We consider that Σ, Δ, and $\mathbb{N}$ are disjoint. However, this does not reduce the generality of our constructions, since one can easily add some special delimiters in the proper places. Nonetheless, the presence of such delimiters will diminish the clarity of the constructions, hence we will omit them.

4.3.1 Computing with Deadlines

One of the most often encountered real–time features is the presence of *deadlines*. The deadlines are typically classified into *firm* deadlines, when a computation that exceeds the deadline is useless, and *soft* deadlines, where the usefulness of the computation decreases as time elapses [16].

For example, a firm deadline may be expressed as "this transaction must terminate within 20 seconds from its initiation". By contrast, a soft deadline may be "the usefulness of this transaction is max before 20 seconds elapsed; after this deadline, the usefulness is given by the function $u(t) = max \times 1/(t-20)$".

Let Π be a problem whose instances can be classified into three classes: (i) no deadline is imposed on the computation; (ii) a firm deadline is imposed at time t_d; (iii) a soft deadline is imposed at time t_d, and the usefulness function is u after this deadline, $u : [t_d, \infty) \to \mathbb{N} \cap [max, 0]$. We build for each instance a timed ω-word (σ, τ) over $\Sigma \cup \Delta \cup (\mathbb{N} \cap [max, 0]) \cup \{w, d\}$, $w, d \notin \Sigma \cup \Delta$ as follows:

(i) $\sigma_1 \ldots \sigma_m = o$, $\sigma_{m+1} \ldots \sigma_{m+n} = \iota$, $\sigma_i = w$ for $i > m+n$, $\tau_i = 0$ for $1 \leq i \leq m+n$, and $\tau_i = i - m - n$ for $i > m+n$.

(ii) $\sigma_1 \in \mathbb{N} \cap [max, 0)$, $\sigma_2 \ldots \sigma_{m+1} = o$, $\sigma_{m+2} \ldots \sigma_{m+n+1} = \iota$, $\tau_i = 0$ for $1 \leq i \leq m+n+1$; if $\tau_i < t_d$ and $i > m+n+1$, then $\tau_i = i - m - n - 1$ and $\sigma_i = w$. Let i_0 be the index such that $\tau_i = t_d$. Then, for all $i \geq i_0$, $\tau_i = i_0 + \lfloor (i - i_0)/2 \rfloor$, and

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

(iii) This case is the same as case (ii), except that equation (3) becomes

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise.} \end{cases} \tag{4}$$

Let the language formed by all the ω-words that conforms to the above description be L. Basically, a timed ω-word in L has the following properties: At time 0, a possible output and a possible input for Π are available. Then, up to the deadline d, the symbols that arrive are w. After that, each time unit brings to the input a pair of symbols, the first component being d (signaling that the deadline passed), and the second one being the measure of usefulness the computation still has (which is 0 for ever when the deadline is firm). When a deadline is imposed over the computation (cases (ii) and (iii)), a minimum acceptable usefulness estimate is also present at the beginning of the computation. Let then $L(\Pi)$ be the language of successful instances of Π, $L(\Pi) \subseteq L$, in the sense that, an ω-word x from L is in $L(\Pi)$ iff some algorithm that solves Π, when processing the input from x, outputs the output from x either within the imposed deadline (if any), or at a time when the usefulness of the process is not below the acceptable limit from x.

We are ready to present now an acceptor for $L(\Pi)$. For simplicity, we consider that this acceptor is composed of two "processes", P_w and P_m. P_w is an algorithm that solves Π, which works on the input of Π contained in the current input ω-word, and stores the solution in some designated memory space upon termination. If there is more than one solution for the current instance, then P_w nondeterministically chooses that solution that matches the proposed solution contained in the ω-word, if such a solution exists. Meantime, P_m monitors the input. If, at the moment P_w terminates, the current symbol is w, then P_m compares the solution computed by P_w with the proposed solution, and imposes to the whole acceptor the "final" state f if they are identical, or some other designated state r (for "reject") otherwise.

On the other hand, if at the moment P_w terminates, the current symbol is d, then the deadline passed. Then, P_m compares the current usefulness measure with the minimum acceptable one. If the usefulness is not acceptable, then P_m imposes the state r on the whole acceptor. Otherwise, P_m compares the result computed by P_w with the proposed solution, and imposes either the state f or r, accordingly.

Once in one of the states f or r, the acceptor keeps cycling in the same state.

It is immediate that the language accepted by the above acceptor is exactly $L(\Pi)$, and hence we completed the modeling of computations with deadlines in terms of ω-languages. Note that we assumed here that all the input data are available at the beginning of computation. However, the case when data arrive while the computation is in progress is easily modeled by modifying the timestamps that corresponds with each input data. But this case is covered in more details by our discussion in section 4.3.2.

4.3.2 The Data–Accumulating Paradigm

The data–accumulating paradigm has been extensively studied in [9, 10, 19, 20]. A *data–accumulating algorithm* (or *d-algorithm* for short) works on an input considered as a virtually endless stream. The computation terminates when all the currently arrived data have been processed before another datum

arrives. In addition, the arrival rate of the input data is given by some function $f(n,t)$ (called the *data–arrival law*), where n denotes the amount of data that is available beforehand, and t denotes the time. The family of arrival laws most commonly used as examples is

$$f(n,t) = n + kn^{\gamma}t^{\beta}, \tag{5}$$

where k, γ, and β are positive constants. A successful computation of a d-algorithm terminates in finite time.

Given a problem Π pertaining to this paradigm, we can build the corresponding timed ω-language $L(\Pi)$ similarly to section 4.3.1. More precisely, given some (infinite) input word ι for Π (together with a data arrival law $f(n,t)$ and an initial amount of data n), and a possible output o of an algorithm solving Π with input ι, a timed ω-word (σ,τ) that may pertain to $L(\Pi)$ is constructed as follows: $\sigma_1 \ldots \sigma_m = o$, $\sigma_{m+1} \ldots \sigma_{m+n} = \iota_1 \ldots \iota_n$, $\tau_i = 0$ for $1 \leq i \leq m+n$. Note that, since both the arrival law and the initial amount of data are known, one can establish the time of arrival for each input symbol ι_j, $j > n$. Let us denote this arrival time by t_j. Also, let $i_0 = m+n+1$. Then, the continuation of the timed ω-word is as follows: for all $i \geq 0$, $\sigma_{i_0+2i} = c$ (where c is a special symbol), and $\sigma_{i_0+2i+1} = \iota_{i_0+i}$; moreover, $\tau_{i_0+2i+1} = t_{i_0+i}$, and $\tau_{i_0+2i} = \tau_{i_0+2i+1} - \epsilon$, where ϵ is a constant infinitesimally close to 0.

Now, an acceptor for $L(\Pi)$ has a structure which is identical[1] to the one used in section 4.3.1. More precisely, it consists in the two processes P_w and P_m. P_w works exactly as the P_w from section 4.3.1, except that it emits some special signal to P_m each time it finishes the processing of one input data. Note that, since any d-algorithm is an on–line algorithm [10], it follows that, once such a signal is emitted the p-th time, P_w has a (partial) solution immediately available for the input word $\iota_1 \ldots \iota_p$.

Then, suppose that P_m received p signals from P_w, and it also received the input symbol $\sigma_{i_0+2(p-1-i_0)}$, but it didn't receive yet the input symbol $\sigma_{i_0+2(p-i_0)}$. This is the only case when P_m attempts to interfere with the computation of P_w. In this case, P_m compares the current solution computed by P_w with the solution proposed in the input ω-word; if they are identical, the input is accepted, and the input is rejected otherwise (in the sense that either state f or r is imposed upon the acceptor, accordingly).

Again, once in one of the states f or r, the acceptor keeps cycling in the same state. It is immediate that $L(\Pi)$ contains exactly all the successful instances of Π, therefore we succeeded in modeling d-algorithms using timed ω-languages.

Other related paradigms, like c-algorithms [11, 19, 20] (which are similar with d-algorithms, except that data that arrive during the computation consist in corrections to the initial input rather than new input) can be easily modeled using the same technique.

[1] In particular, if there is more than one solution for the current instance, then P_w nondeterministically chooses that solution that matches the proposed solution contained in the ω-word, if such a solution exists.

We modeled in sections 4.3.1 and 4.3.2 the two main ingredients that, when present, impose the real–time qualifier on the problem. This supports our thesis that the theory of timed languages covers all the practically relevant aspects of real–time computations.

Note that, even if we considered here only ω-languages, finite timed languages may be a useful tool too. For illustrating this, let us get back to the language L_ω from the proof of theorem 4.1. Here, one can notice that, even if the words from the language themselves are infinite, by analyzing portions only (more precisely, those portions delimited by \$ symbols), one can draw conclusions regarding the phenomena that take place. However, there are real–time problems that probably cannot be modeled as finite timed languages. Take for example the theory of d-algorithms where, although any successful computation considers only a finite amount of input data, the input itself is infinite.

Some authors include *reactive algorithms* as a special class of real–time algorithms. In this view a reactive algorithm is required to respond to the input without breaking some fixed deadline. This case is obviously covered by the definition we proposed. However, other papers relax this condition [12]. They continue to ask that the algorithm responds before some deadline, but this deadline is not fixed anymore, it being, for example, a function of the length of the input. Since we didn't constrain the time sequence associated to a timed word (except for monotonicity and progress conditions), this paradigm can be easily expressed in terms of timed languages.

5 Conclusions, or Towards a Complexity Theory of Real–Time Computations

We believe that the notion of timed languages and acceptors as introduced in section 4 are important tools in developing a complexity theory for real–time systems, which is simply not present at this time. We presented in this paper a general definition of this class of languages, and we suggested that this definition is powerful enough to model all the practically important aspects of real–time computations. We also supported our thesis with meaningful examples.

As a first step toward the goal of real–time complexity theory, one can study the hierarchy of timed languages, similar with the Chomsky hierarchy for normal languages, together with the closure properties and with the corresponding classes of acceptors. This direction has been initiated in [8], with the study of timed ω-automata, and we suggested in this paper a general form of such an acceptor.

Nonetheless, we believe that the most interesting direction is the establishment of a complexity theory for real–time systems, based on the definition of timed languages. In general, such a theory takes into account the measurable resources used by an algorithm, most important being the time and the space. However, in the real–time environment, time complexity makes little sense, since in most applications the time properties are established beforehand. But, as

supercomputing is now a reality, a complexity hierarchy with respect to the number of processors is a very interesting direction, with promising prospects. Note that it has been already established that a parallel approach can make the difference between success and failure [6, 9, 10, 11, 20], or can enhance significantly the quality of solutions [2, 3, 4, 5].

Note that a similar research was started in [21, 23], where the hierarchy was established with respect to the number of tapes of real–time Turing machines. However, on one hand, a multitape Turing machine is probably not equivalent to a multiprocessor device, and, on the other hand, since the real–time domain is a highly practical issue, we think that the use of models closer to real machines (e.g., the PRAM [1]) is desirable.

References

[1] S. G. Akl, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, NJ, 1997.

[2] ——, *Secure file transfer: A computational analog to the furniture moving paradigm*, Tech. Rep. 99-422, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1999. http://www.qucis.queensu.ca/~akl/techreports/furniture.ps.

[3] S. G. Akl and S. D. Bruda, *Real-time optimization: Beyond speedup*, to appear in: Parallel Processing Letters, 9 (1999). For a preliminary version see http://www.cs.queensu.ca/~akl/techreports/beyond.ps.

[4] ——, *Real-time cryptography: Beyond speedup II*, Tech. Rep. 99-423, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1999. http://www.cs.queensu.ca/~akl/techreports/realcrypto.ps.

[5] ——, *Real-time numerical computation: Beyond speedup III*, Tech. Rep. 99-424, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1999. http://www.cs.queensu.ca/~akl/techreports/realnum.ps.

[6] S. G. Akl. and L. Fava Lindon, *Paradigms admitting superunitary behaviour in parallel computation*, Parallel Algorithms and Applications, 11 (1997), pp. 129–153.

[7] W. Albrecht and D. Zöbel, *Integrating fixed priority and static scheduling for external consistency*, in Real–Time Databases and Information Systems: Recent Advances, A. Bestavros and V. Fay-Wolfe, eds., Kluwer Academic Publishers, 1997, pp. 89–104.

[8] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science, 126 (1994), pp. 183–235.

[9] S. D. Bruda and S. G. Akl, *On the data-accumulating paradigm*, in Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, NC, October 1998, pp. 150–153. For a preliminary version see http://www.cs.queensu.ca/~bruda/www/data_accum.

[10] ——, *The characterization of data-accumulating algorithms*, in Proceedings of the International Parallel Processing Symposium, San Juan, Puerto Rico, 1999, pp. 2–6. For a preliminary version see http://www.cs.queensu.ca/~bruda/www/data_accum2.

[11] ——, *A case study in real-time parallel computation: Correcting algorithms*, in Proceedings of the Midwest Workshop on Parallel Processing, Kent, OH, 1999. For a preliminary version see http://www.cs.queensu.ca/~bruda/www/c-algorithms.

[12] E. A. Emerson, *Real–time and the Mu-calculus (preliminary report)*, in Real–Time: Theory in Practice, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, eds., 1991, pp. 176–194. Springer Lecture Notes in Computer Science 600.

[13] P. C. Fischer, *Turing machines with a schedule to keep*, Information and control, 11 (1967), pp. 138–146.

[14] C.-C. Han and H.-Y. Tyan, *A better polynomial–time schedulability test for real–time fixed–priority scheduling algorithms*, in Proceedings of the 18th IEEE Real–Time Systems Symposium, San Francisco, CA, Dec 1997, pp. 36–45.

[15] K. Jeffay, *The real-time producer/consumer paradigm a paradigm for the construction of efficient, predictable real-time systems*, in Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, 1993, pp. 796–804.

[16] M. R. Lehr, Y.-K. Kim, and S. H. Son, *Managing contention and timing constraints in a real–time database system*, in Proceedings of the 16th IEEE Real–Time Systems Symposium, Pisa, Italy, Dec 1995, pp. 332–341.

[17] N. Leveson and J. Stolzy, *Analyzing safety and fault tolerance using timed Petri nets*, in Proceedings of the International Joint Conference on Theory and Practice of Software Development, 1985, pp. 339–355. Springer Lecture Notes in Computer Science 186.

[18] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[19] F. Luccio and L. Pagli, *The p-shovelers problem (computing with time-varying data)*, in Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing, 1992, pp. 188–193.

[20] ——, *Computing with time–varying data: Sequential complexity and parallel speed–up*, Theory of Computing Systems, 31 (1998), pp. 5–26.

[21] M. O. Rabin, *Real time computations*, Israel Journal of Mathematics, 1 (1963), pp. 203–211.

[22] S. Rao Kosaraju, *Real–time pattern matching and quasi–real–time construction of suffix trees (preliminary version)*, in Proceedings of STOC 94, Montreal, Quebec, Canada, May 1994, pp. 310–316.

[23] A. L. Rosenberg, *Real-time definable languages*, Journal of the ACM, 14 (1967), pp. 645–662.

[24] M. Thorin, *Real–Time Transaction Processing*, Macmillan, Hampshire, UK, 1992.

[25] H. Yamada, *Real-time computation and recursive functions not real-time computable*, IRE Transactions on Electronic Computers, EC-11 (1962), pp. 753–760.