Technical Report No. 2000-435

Real-Time Computation: A Formal Definition and its Applications*

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

February 8, 2000

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 present some important features of real-time algorithms, namely the presence of deadlines and the continuous, real-time, arrival of input data, together with their corresponding models in our formalism. Moreover, in order to illustrate the expressive power of our formalism, we also model aspects from the practical areas of real-time database systems and ad hoc networks. We believe that the proposed model 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, real-time databases, ad hoc networks.

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.

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 [39] and further studied in [22, 34, 36]. 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 [26], 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 [11]. 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 [11] 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)

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

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 [11] 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 [11], 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. Then, we direct our attention to two larger practical areas of real-time computations, modeling important problem from real-time database systems and ad hoc networks using the proposed formalism. We 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 some important features of real-time algorithms, together with their models in our formalism in sections 5, 6, and 7. 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 a 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 [29]. For a k-tape Turing machine M, we denote a configuration of M by a tuple $(q, l_1\underline{a_1}r_1, \ldots, l_k\underline{a_k}r_k)$, where q is the current state, and for all i, $1 \leq i \leq k$, $l_ia_ir_i$ is the current content of tape i, and a_i is the symbol contained in the tape cell currently read by the head of tape i. 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 $\vdash_M$. We denote by $\#$, h, and λ the blank symbol, halt state, and empty word, respectively.

3 Previous Work

3.1 Real-Time Turing Machines

Probably the first work on formalizing the notion of real-time is [39]. 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 [34, 36].

Definition 3.1 [36]

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 distinct 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 to 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

$$\begin{aligned}(q, u\underline{a}bv, x_1, \ldots, x_k) &\vdash_M (q', ua\underline{b}v, x'_1, \ldots, x'_k),\\ (q'', u\underline{a}bv, x_1, \ldots, x_k) &\vdash_M (q', u\underline{a}bv, x'_1, \ldots, x'_k),\\ (q, u\underline{\#}\lambda, x_1, \ldots, x_k) &\vdash_M (h, u\underline{\#}\lambda, x'_1, \ldots, x'_k),\end{aligned}$$

for some words x_i, x'_i over the working alphabet of M, $1 \leq i \leq k$, and $a \neq \#$. In other words, M has to consume an input symbol each time it is in a polling state, and is not allowed to move the head on the input tape while in an autonomous state.

M accepts the input $w = aw'$ iff $(s, \underline{a}w', x_1, \ldots, x_k) \vdash^*_M (h, w\underline{\#}, x'_1, \ldots, x'_k)$, where s is the initial state.

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 [22], 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 [35]. 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 [10, 37], real-time scheduling [24], tracking devices [26], or process control [28], 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 [28], 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 [26]. 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 [26] 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 5.2), where the arrival rate varies over time.

3.3 Timed Automata

Finally, a third model of real-time computation is the *timed (finite) automaton* [11]. 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 [11] 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

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

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 Muller 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 [11] 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 [11] 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 [11] (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 \$. Note that $S_1, S_2 \subseteq S$, hence both S_1 and S_2 are finite. 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 some time sequence, and call the language obtained in this way L'_ω. Then, the proof by contradiction follows from the proof of theorem 4.1. Indeed, consider a TBA that is identical to A' from the mentioned proof, and for which $C = \emptyset$. Clearly, this TBA recognizes L'_ω. However, such an automaton is an impossibility. □

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 [11], 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 $|\sigma| + 1 = |\tau|$.

   The time value $\tau_\#$ associated to the symbol # denote the time at which the recognition process of the timed word must terminate.

   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 [11], except that we consider finite timed languages as well. However, while the study in [11] 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 light of the above definition, we can also establish the general form of an acceptor for timed languages. Extending the idea from [11], 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

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

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, although a clock may be reset only at the time some input symbol is read.

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*.

Definition 4.2 A timed acceptor $\mathcal{A}$ 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 ω-timed 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 $(\sigma, \tau) \in L'$. □

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 its 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 (see, for example, [25] and the references therein). We already presented the definition from [36], which is given terms of Turing machines, but it can be easily extended to other models (definition 3.1, item 1).

Generally, 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.

On the other hand, 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 [39] 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 Operations on Timed languages

The union, intersection, and complement are straightforwardly defined. Moreover, it is immediate that the language that results from such an operation on two timed languages is a timed language as well. On the other hand, the concatenation is a more complex issue. Indeed, the naive operation of concatenation of two (finite) timed words (that simply concatenates together the pair of sequences of symbols and the pair of time sequences) fails to produce a timed word, since the result of the time sequence concatenation is likely not a time sequence. This naive approach is even worse in the case of ω-words, where concatenating two infinite sequences makes little sense.

However, one can rely on the semantics of timed words in defining a meaningful concatenation operation. More precisely, recall that a timed word means a sequence of symbols, where each symbol has associated a time value that represents the moment in time when the corresponding symbol becomes available. Then, it seems natural to define the concatenation of two timed words as the union of their sequences of symbols, ordered in nondecreasing order of their arrival time. Formally, we have the following definitions.

Definition 4.3 Given two (infinite or finite) words $\sigma = \sigma_1\sigma_2,\ldots$ and $\sigma' = \sigma'_1\sigma'_2,\ldots$, we say that σ' is a *subsequence* of σ iff (a) for each positive integer i there is a positive integer j such that $\sigma'_i = \sigma_j$, and (b) for any positive integers i, j, k, l such that $\sigma'_i = \sigma_j$ and $\sigma'_k = \sigma_l$, $i > k$ iff $j > l$. □

Definition 4.4 Given some alphabet Σ, let (σ', τ') and (σ'', τ'') be two timed ω-words over Σ. Then, the we say that (σ, τ) is the *concatenation* of (σ', τ') and (σ'', τ''), and we write $(\sigma, \tau) = (\sigma', \tau')(\sigma'', \tau'')$, iff

1. both $(\sigma'_1, \tau'_1)(\sigma'_2, \tau'_2)\ldots$ and $(\sigma''_1, \tau''_1)(\sigma''_2, \tau''_2)\ldots$ are subsequences of $(\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$,
2. for any $d \in \{', ''\}$ and any positive integers i and j, $i < j$, such that $\tau^d_k = \tau^d_l$ for any k, l, $i \le k < l \le j$, there exists m such that, for any $0 \le \iota \le j - i$, $(\sigma_{m+\iota}, \tau_{m+\iota}) = (\sigma^d_{i+\iota}, \tau^d_{i+\iota})$, and
3. for any positive integers i and j such that $\tau'_i = \tau''_j$, there exists k and l, $k < l$, such that $(\sigma_k, \tau_k) = (\sigma'_i, \tau'_i)$ and $(\sigma_l, \tau_l) = (\sigma'_j, \tau'_j)$.

Given two timed ω-languages L_1 and L_2, the concatenation of L_1 and L_2 is the timed ω-language $L = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$. The concatenation of two (finite) timed words and two timed languages is defined analogously. □

In addition to the mentioned order of the resulting sequence of symbols, two more constraints are imposed in definition 4.4. These constraints order the result in the absence of any ordering based on the arrival time. First, if either of the two ω-words contain some subword of symbols that arrive at the same time, then this subword is a subword of the result as well. That is, the order of many symbols that arrive at the same time is preserved. Then, if some symbols σ_1 and σ_2 from the two ω-words that are to be concatenated, respectively, arrive at the same moment, then we ask that σ_1 precedes σ_2 in the resulting ω-word. The concept of Kleene closure for timed languages can be then defined based on the concatenation operation:

Definition 4.5 Given some timed (ω-)language L, let $L^1 = L$, and, for any fixed $k > 1$, $L^k = LL^{k-1}$. Furthermore, let $L^* = \cup_{k>0} L^k$. We call L^* the Kleene closure of L. □

It is clear that the set of timed (ω-)languages is not closed under Kleene closure. Indeed, it is immediate that there are words in L^* whose time sequence does not satisfy the progress condition. In order to summarize the properties of timed languages, we state the following result.

Theorem 4.3 *The set of (ω-)languages is closed under intersection, union, complement and concatenation, under a proper definition of the latter. However, it is not closed under Kleene closure.* □

5 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 $\tau_{i+1} - \tau_i$ 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 [26]), 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.

5.1 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 [27].

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 $1 \le i \le m+n$, 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 $1 \le i \le m+n+1$; 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 $i \ge i_0$, $\tau_i = i_0 + \lfloor (i - i_0)/2 \rfloor$, and

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

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

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

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 5.2.

5.2 The Data Accumulating Paradigm

The data accumulating paradigm has been extensively studied in [15, 16, 30, 31]. 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 5.1. 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 5.1. More precisely, it consists in the two processes P_w and P_m. P_w works exactly as the P_w from section 5.1, except that it emits some special signal to P_m each time it finishes the processing of one input data. Note that,

[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.

since any d-algorithm is an on-line algorithm [16], 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 $\iota_1 \dots \iota_p$.

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 [17, 30, 31] (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.

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 [21]. 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.

6 Real-Time Database Systems and Timed Languages

We modeled in sections 5.1 and 5.2 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. However, another part of this thesis is that our model is capable of capturing practical aspects of the real-time domain. In order to further emphasize this aspect, we provide in what follows timed ω-languages that model problems from two highly practical areas, namely real-time databases and ad hoc networks.

6.1 Relational Database Systems

Much of this section is presented conforming to [2]. Throughout the paper we consistently use the notations from [2]. It is assumed that a countably infinite set **att** of attributes is fixed. Moreover, the countably infinite set **dom** (disjoint from **att**) is also fixed, and it represents the underlying domain. A *constant* is an element of **dom**. When different attributes should have different domains, a mapping *Dom* on **att** is considered, where $Dom(A)$ is a set called the domain of A, $Dom(A) \subseteq \mathbf{dom}$. There is a countably infinite set of relation names. A relation is given by its name and its ordered set of attributes (sometimes called its *sort*). Given a relation R, the sort of R is denoted by $sort(R)$, and the arity of R is defined as $arity(R) = |sort(R)|$. A *relation schema* is a relation name R. A *database schema* is a nonempty finite set $\mathbf{R}$ of relation names. Let R be a relation or arity n. Then, a *tuple* over R is an expression $R(a_1, a_2, \dots, a_n)$, where $a_i \in \mathbf{dom}$, $1 \leq i \leq n$. A *relation instance* over R is a finite set of tuples over R. Given a database schema $\mathbf{R}$, a *(database) instance* $\mathbf{I}$ over $\mathbf{R}$ is a finite set that is the union of relation instances over R, for $R \in \mathbf{R}$. The sets of instances over a database schema $\mathbf{R}$ and a relation schema R are denoted by $inst(\mathbf{R})$ and $inst(R)$, respectively.

	Title	*Description*	*Artist*
Exhibitions	Terre Sauvage	Canadian Landscape Paintings	Thompson
	Terre Sauvage	Canadian Landscape Paintings	Harris
	Terre Sauvage	Canadian Landscape Paintings	MacDonald
	Painter of the Soil	Works on Paper	Schaefer
	Sorrowful Images	Early Nederlandish Devotional Diptychs	Aelbrecht
	Sorrowful Images	Early Nederlandish Devotional Diptychs	Dieric

	City	*Title*	*Date*
Schedules	Mexico City	Terre Sauvage	October 1999
	St. Catharines	Painter of the Soil	November 1999
	Hamilton	Sorrowful Images	November 1999

Figure 1: An example of a relational database instance.

	Artist	*City*
S	Schaefer	St. Catharines
	Aelbrecht	Hamilton
	Dieric	Hamilton

Figure 2: The result of a query.

In order to support the intuition behind the concepts presented above, let us consider as an example the relational database[2] from figure 1.

The database schema (call it NGC) for the database shown in figure 1 is defined by $NGC = \{Exhibitions, Schedules\}$. It contains therefore two relation schemae, namely *Exhibitions* and *Schedules*. The attributes *Title*, *Description*, *Artist*, *City*, and *Date* are included in the set **att**. One can consider the set of finite length strings of characters as the underlying domain **dom**. However, a mapping *Dom* on **att** may be considered as well. In this example, $Dom(Title) \supseteq \{\text{Terre Sauvage, Painter of the Soil, Sorrowful Images}\}$, and so on. Furthermore, $sort(Exhibitions) = \{Title, Description, Artist\}$, and therefore $arity(Exhibitions) = 3$. Finally, the relation instance over *Exhibitions* from figure 1 contains 6 tuples, while the instance over *Schedules* contains only 3 tuples.

The interrogation of a database is accomplished by using *queries*. A query is a partial mapping from $inst(\mathbf{R})$ to $inst(S)$, for a fixed database schema $\mathbf{R}$ and a fixed relation schema S.

For example, a meaningful query (expressed in plain English) for the database from figure 1 might be "which artist is exhibited in which city in November." This query is a map from $inst(NGC)$ to $inst(S)$ for some relation schema S, where $sort(S) = \{Artist, City\}$. Incidentally, the result of performing this query on the database instance from figure 1 is shown in figure 2.

6.1.1 Complexity of queries

We are mainly concerned with *data complexity* of queries, namely the complexity of evaluating a fixed query for variable database inputs [2], since the usual situation is that the size of the database input dominates by far the size of the query (and therefore this measure is most relevant).

The complexity of queries is defined based on the *recognition problem* associated with the query. More precisely, for a query q, the recognition problem is: Given an instance $\mathbf{I}$ and a tuple u, determine if u belongs to the answer $q(\mathbf{I})$. That is, the recognition problem of a query q is the language

[2]The events described in this example are loosely based on [1].

$$\{enc(\mathbf{I})\$enc(u)|u \in q(\mathbf{I})\}, \tag{6}$$

where *enc* denotes a suitable encoding over queries and tuples, and \$ is a special symbol.

The *(data) complexity* of q is the (conventional) complexity of its recognition problem. Then, for each conventional (time, space, processors) complexity class C, one can define a corresponding *complexity class of queries* QC.

Another way to define the complexity of queries is based on the complexity of actually constructing the result of the query. The two definitions are in most cases interchangeable [2].

6.2 Real-Time Database Systems

The theory of real-time database systems (RTDBSs) may be viewed as the meeting point for the areas of active databases, and temporal databases. In the following, we make a minimal presentation of this theory, directing the interested reader to [2, 32].

6.2.1 Active databases

Active databases support the automatic triggering of updates in response to (internal or external) events. Forward chaining of *rules* is generally used to accomplish the response, as in the case of expert systems. The component of active database is central in the theory of real-time databases, since these databases usually have to respond in a timely fashion to changes in the environment, that are usually signalled to the database system by events.

There are three components in an active database [2]: an event monitoring subsystem, a set of rules (often called a *rule base*), and a semantics for rule application (or an *execution model*).

The typical form of a rule is "**on** *event* **if** *condition* **then** *action*," where the *event* may be either an external phenomenon, or an internal event (such as the insertion of a tuple). Events may have attributes that are passed to the system. The conditions may involve parameters that are passed along with the event, or parameters that are specific to the content of the database. The action is an arbitrary routine, that usually involves an updating transaction. An action may in turn generate other events and hence trigger other rules.

For example, we may want to consider deleting those exhibitions contained in the instance of relation *Schedules* from the database shown in figure 1 which are no longer exhibited in the specified city. A rule for such a processing might be

$$\textbf{on } \mathit{MonthChange} \textbf{ if } \mathit{true} \textbf{ then } \mathit{del}(\mathit{Date} < \mathit{CurrentDate}),$$

where $del(C)$ deletes those tuples for which condition C holds.

A fundamental issue in active databases addresses the choice of an execution model, that specifies how and when rules are applied. An important dimension of variation between execution models is given by the moment the rules are fired. The first model is *immediate firing*, where a rule is fired as soon as its event and condition become true. Under *deferred* firing, rule invocation is delayed until the final state (in the absence of any rule) is reached. Finally, when a separate process is spawned for the rule action and is executed concurrently with other processes, we have an *concurrent* firing. In the most general model, each rule have an associated firing mode (immediate, deferred, concurrent).

Besides the firing mode, there is a wide variety of execution models. A dimension of flexibility that is of interest in the area of real-time databases concerns the access to the "past" of a database. That is, in addition to the access of the current state, a rule may have access to one or more previous states. In a real-time environment, there is usually a need to have full access to (a part of) the history of the system, as we shall see in section 6.2.3.

6.2.2 Temporal databases

Classical databases model static aspects of the data. However, in many applications (and especially in the real-time case), the history of data is just as important as the data itself. Let our point of interest be a database over some schema $\mathbf{R}$, and let us consider now the content of the database through time. Basically, one can associate to each time t the instance $\mathbf{I}_t$ of the database at time t. That is, the database appears as a sequence of *states* or *snapshots* indexed by some time domain. In order to query temporal databases, relational languages must be extended to take into account the time dimension. To say that a tuple u is in relation R at time t, one could simply add a second argument to R and write $R(u,t)$.

However, an important issue concerns the time domain [2, 32]. First of all, the *structure* of time should be addressed. There are two structural models of time, *linear* and *branching*. In the latter model, time is linear and totally ordered from past till "now", when it divides into several time lines. The main model for real-time databases is, however, linear. The *density* of the time domain is also of interest. This domain may be either *continuous* (isomorphic to reals), dense (isomorphic to rationals), or discrete (isomorphic to natural numbers). The model of choice is usually the discrete time domain, where each natural number corresponds to a nondecomposable unit of time, sometimes referred to as a *chronon*. Finally, one can differentiate between *relative* and *absolute* time.

The *dimensionality* of time addresses the question "what is the meaning of $\mathbf{I}_t$." In this respect, one can differentiate between *valid time* (the time associated to each object in some database instance is the time at which the fact associated to this object became true in reality) and *transaction time* (the time at which the fact was recorded in the database as stored data).

Although the concept of a temporal database as a sequence of instances in very convenient theoretically, this is an extremely inefficient way to represent such databases. In practice, this information is summarized in a single database, by using *timestamps* to indicate the time of validity. Such timestamps may be placed at attribute or tuple level (generally, because of this alternative, we use the term "object" when referring to either an attribute or a tuple), and are typically unions of intervals over the temporal domain [2, 32, 38].

6.2.3 Real-Time Databases

Real-time databases combine the notions of active and temporal databases. There are therefore two important aspects: First, a real-time database interacts with the physical world, for example by reading values of physical objects and storing them. The real world is periodically sampled, and each such sampling process generates an event that must be handled by the database. In addition, when a value changes, some related changes happen to other data. This update process is typically accomplished by rule application as well. Needless to say, since data in a real-time database is time sensitive, such a database is a temporal one. Second, the transactions must be timely, that is, they must complete within their time constraints (deadlines).

We briefly present in the following the data model used in [38], which derives from the historical relational data model [19].

The objects from the database are grouped in three categories: *Image objects* are those objects that contain information that is obtained directly from the external environment. Associated with an image object is the most recent sampling time. Archival sets of image objects are typically maintained, so that different snapshots at different points in time are available. A *derived object* is computed from a set of image objects and possibly other objects. The timestamp associated with a derived object is the oldest valid time of the data objects used to derive it. Finally, an *invariant object* is a value that is constant with time. Such an object may be considered either a temporal or non temporal data. In the first case, the timestamp associated with such an object is always the current time. Note that this is a natural classification. Moreover, in order to keep a complete history of the database, it is enough to keep archival copies of the image objects, since the other objects are either invariant with time, or their values can be derived from the values of the other objects.

The time associated with some object x is denoted by t_x. It is assumed for now that the time of an object is a single point in time.

The age $a(x)$ of an object x is the difference between the current time and the timestamp of x. The dispersion of two data objects x and y is the absolute value of the difference between the timestamps of x and y. Given some set of objects Y, it is absolutely consistent if $a(x_i) \leq T_a$, for all $x_i \in Y$, and where T_a is some specified (fixed) threshold. Similarly, Y is relatively consistent if $d(x_i, x_j) \leq T_r$ for all $x_i, x_j \in Y$, where T_r is another specified threshold.

Then, a real-time database instance is defined as $B = (I_1, I_2, ..., I_n, D, V)$, where I_n is the most recent set of image objects, and $I_1, I_2, \ldots, I_{n-1}$ are archival variants of this set. D is the set of derived objects, and V

is the set of invariant ones. The database is said to have absolute consistency if I_n is absolutely consistent and the ages of data objects used to derive the derived objects are less than the specified threshold. The conditions for relative consistency are similar.

Since in a real-time database it is important to reflect the state of the real world, it is assumed that the difference between the valid time and the transaction time is small. Time is usually considered discrete. The valid time associated with each temporal object in the database instance is called the *lifespan* of the object. The lifespan of a data object is defined as a finite union of intervals. These intervals are closed under union, intersection and complementation, and form therefore a boolean algebra. A single instance of time is represented by a degenerated interval that contains exactly one time value. A lifespan can also be associated with a set of objects, in a natural manner.

Based on these notions, a variant of relational algebra is defined as a query language for real-time databases in [19, 38].

Additional issues in the real-time database systems include the pattern of queries (periodic, sporadic, aperiodic), the nature of deadlines (hard, firm, soft) [32], and the way the updating rules are fired. While the first two issues received both theoretical and practical attention in the literature, to our knowledge, there is no special theoretical treatment on the last issue, except for the one that spawn from the active database theory. However, one might study various variants of rule application. For example, one may impose an immediate firing on the rules that update the image objects of the database, but a deferred firing for the derived objects. Note that the immediate firing in the case of image objects is implied in [38] and therefore in the above paragraphs, since it is assumed that the valid and transaction times are close to each other.

Finally, note that some aperiodic query q can be considered as a partial function from B to $inst(S)$, where S is some relation. However, a periodic query returns an answer each time it is issued, therefore such a query is a function from B to $(inst(S))^\omega$.

6.3 Real-Time Database Systems as Timed Languages

As shown in section 6.1.1, one of the ways of assessing the complexity of queries and query languages is based on the reduction of such problems to the problem of language recognition.

However, the real-time component in a real-time database system adds a new dimension to the model, namely the time. Is seems natural therefore to try to model such database systems using timed languages. We describe such a modeling in what follows. We consider that there is a suitable encoding function *enc* that encodes objects and sets of objects, without giving much attention to how such a function is constructed. Note that such functions were widely used (see for example [2, 29]). Let \$ be a symbol that is not in the codomain of *enc*.

Let us ignore the queries for the moment. Recall that a real-time database instance is a tuple $B = (I_1, I_2, ..., I_n, D, V)$, as mentioned in section 6.2.3. Moreover, assume for now that the database contains exactly one immediate object, called o_k, and that the value of o_k is read from the external world each t_k time units. Let D and V be some sets of derived and invariant objects, respectively, with $m = |enc(V)|$ and $p = |enc(D)|$, and $o_k(t)$ be the value of o_k that is read at time t from the external world. Consider then the timed ω-word $db_k = (\sigma, \tau)$, where σ and τ has the following form: let $q = |enc(o_k)|$[3]; then, for every $i \geq 0$, $\sigma_{\alpha+i(q+1)+1} \cdots \sigma_{\alpha+(i+1)(q+1)} = enc(o_k(t_i))\$$,

where $\alpha = m + p + 2$; moreover, $\tau_j = it_i$ for $\alpha + i(q+1) + 1 \leq j \leq \alpha + (i+1)(q+1)$.

Furthermore, let $db_0 = (\sigma, \tau)$, such that $\sigma_1 \ldots \sigma_m = enc(V)$, $\sigma(m+1) = \sigma(m+p+2) = \$$, and $\sigma_{m+2} \ldots \sigma_{m+p+1} = enc(D)$; in addition, $\tau_i = 0$, $1 \leq i \leq m + p + 2$.

In other words, the sets of both invariant and derived objects are specified at time 0, as modeled by the word db_0. Then, each t_k time units a new value for o_k is provided. This is modeled by db_k. It is clear that the database instance is completely specified by the word $db_0 db_k$, since this word models both the invariant and derived objects (by db_0), as well as all the updates for the sole image object (by db_k).

Now, let us consider the general case of a real-time database. That is, we do not restrict ourselves to one invariant object anymore. Therefore, let the database instance contain r such objects, called o_k, $1 \leq k \leq r$.

[3]We assume for the clarity of presentation that the length of the encoding of o_k is constant over time. The extension to a variable length is straightforward.

However, if we consider a word db_k corresponding to each object o_k, $1 \leq k \leq r$, then it is immediate that the database is described by the word

$$db_B = db_0 db_1 \dots db_r. \tag{7}$$

We have now a model for real-time databases. Now, all that we have to do is to consider the queries. Again, we assume without further details that there is a function enc_q for encoding queries and their answers, whose codomain is disjunct from the codomain of enc. Real-time queries can be classified in two classes: periodic and aperiodic [32].

Let us focus on aperiodic queries first. Each such query q may have a firm of soft deadline. However, it seems natural to also consider queries without any deadline, since they might be present even in a real-time environment. Therefore, the encoding of a query should include

1. the time t at which the query is issued,
2. the (encoding of) the query itself $enc_q(q)$,
3. a tuple s that might be included in the answer to the query,
4. the deadline t_d of the query, if any.

Note that a similar problem is the presence of deadlines, that was presented in section 5.1, except that the first item is not modeled (the computation always starts at time 0). Therefore, our construction is similar to the construction of the language that models computations with deadlines.

We have thus a query for which (i) there is no deadline, (ii) a firm deadline is present, or (iii) a soft deadline is present. The deadline (if any) is imposed at some relative time t_d (that is, the moment in time at which the deadline occur is $t + t_d$), and the usefulness function is denoted by u. For each query q and each candidate tuple s we can build similarly to section 5.1 an ω-word $aq_{[q,s,t]} = (\sigma, \tau)$ as follows, where $m = |enc_q(s)\$|$, $n = |enc_q(q)\$|$, and \$, w_q, d_q are not contained in the codomain of enc_q:

(i) $\sigma_1 \dots \sigma_m = enc_q(s)\$$, $\sigma_{m+1} \dots \sigma_{m+n} = enc_q(q)\$$, $\sigma_i = w_q$ for $i > m+n$, $\tau_i = t$ for $1 \leq i \leq m+n$, and $\tau_i = t + i - m - n$ for $i > m + n$.

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

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

(iii) This case is the same as case (ii), except that equation (8) becomes

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise.} \end{cases} \tag{9}$$

Let q be a periodic query now. More precisely, q is issued for the first time at time t, and then it is reissued each t_p time units. Each time q is issued, we have to consider a tuple whose inclusion into the result of q is to be tested. Let s_i be such a tuple for the i-th invocation of q, and let $s = (s_1, s_2, s_3, \dots)$. It is easy to see that such a query is modeled by the word $pq_{[q,s,t,t_p]} = aq_{[q,s_1,t]} aq_{[q,s_2,t+t_p]} aq_{[q,s_3,t+2t_p]} \dots$. However, there is no guarantee that the resulting word $pq_{[q,s,t,t_p]}$ is a timed ω-word. Indeed, the concatenation of an infinite number of timed ω-languages may lead to the violation of the progress rule for the result. In our case, however, the progress condition is not violated, and this follows immediately from the following observation.

Lemma 6.1 *For a word $pq_{[q,s,t,t_p]} = (\sigma, \tau)$, and for any finite positive integer k, there exists a finite integer k' such that $\tau_{k'} \geq k$.*

Proof. Without loss of generality, we assume that $k = t+it_p$ for some $i \geq 0$. However, the symbols for which $\tau_j < k$ can be counted as follows: there are $i+1$ occurrences of some word of the form $enc_q(q)\$enc_q(s_v)\$$, $0 \leq v \leq i$, and at most k occurrences of symbols from $\{w_x, d_x | x = t + lt_p, 0 \leq l \leq i\}$. Therefore, $j \leq (i+1)|enc_q(q)\$enc_q(s)\$| + 2ki$, for some tuple s such that $|s| = \max_{0 \leq v \leq i} s_v$. Clearly, the upper bound for j is finite and therefore so is the number of symbols for which $\tau_j < k$. □

We modeled therefore the main ingredients of a real-time database system. All we have to do then is to put the pieces together.

Definition 6.1 Let B be some real-time database instance. Then, given some aperiodic query q from B to $inst(S)$ (where S is some relation schema), issued at time t, the recognition problem for q on B is the timed ω-language

$$L_{aq} = \{w \in db_B aq_{[q,s,t]} | s \in q(B)\}. \tag{10}$$

Analogously, given a periodic query q from B to $(inst(S))^\omega$, issued at time t and with period t_p, the recognition problem for q on B is the timed ω-language

$$L_{pq} = \{w \in db_B pq_{[q,s,t,t_p]} | s \in q(B)\}. \tag{11}$$

□

Note that the recognition problem for real-time queries is similar to the same problem for conventional queries, shown in relation (6), except that the (conventional) words used in (6) are replaced by timed ω-words.

7 Ad Hoc Networks

An *ad hoc network* is a collection of wireless mobile *nodes*, that dynamically forms a temporary network without using any existing network infrastructure or centralized administration [13, 23]. Due to the limited transmission range of such nodes, multiple hops may be needed for one node to exchange data with another.

The main difference between an ad hoc network and a conventional one is the routing protocol. In such a network, each host is mobile. Therefore, the set of those nodes that can be directly reached by some host changes with time. Furthermore, because of this volatility of the set of directly reachable nodes, each mobile node should act not only as a host, but as a router as well, forwarding packets to other mobile hosts in the network.

Although the concept of ad hoc networks is relatively new, many routing algorithms were developed (see, for example, [12, 13] and the references therein). However, little in known about the performances of these algorithms. A comparative performance evaluation was proposed for the first time in [13], where several routing algorithms are compared based on discrete event simulation. To our knowledge, no analytical model have been proposed up to date.

On the other hand, an ad hoc network is obviously a real-time system. Indeed, since the positions (and implicitly the connectivity) of all the hosts are functions of time, such a network is close to the correcting algorithms paradigm [17]. Therefore, conforming to our claim that timed languages can model all the meaningful aspects of real-time computations, one can model ad hoc networks using this formalism. This is what we are attempting in the following.

7.1 Assumptions and Notations

When speaking about ad hoc networks, we assume that the maximum number of nodes in such a network is upper bounded by a finite constant. We believe that this is a reasonable assumption, given today's bandwidth limits and the potential uses of such networks. We also assume that, if a message is emitted by some node at some time t and received by another node that is in the transmission range of the sender at time t', then $t' = t+1$. That is, transmitting a message takes one time unit. Note that we actually established in this way a granularity of the time domain. This granularity seems appropriate, since transmitting a message is an elementary operation in a network.

Finally, we introduce a notation for the transmission range. We denote this characteristic by the predicate $range(n_1, n_2, t)$. That is, a node n_2 is in the transmission range of other node n_1 at time t iff $range(n_1, n_2, t) = true$. We do not give any specific way of computing this predicate, since such a computation depends on the characteristics of the particular application. Indeed, this predicate depends on the characteristics of both n_1 and n_2, as well as on the geographical characteristic of the area between the two nodes.

7.2 Nodes as Timed ω-Words

The main component of a model for ad hoc networks is the mobile host (or the node). It is consistent to assume that each node in a network is uniquely identified (for example, by its unique IP address). For convenience, we label such a node by an integer between 1 and n, where n is the number of nodes in the given network.

We assume that there is an encoding function e of the properties of any node i (like the label i of the node, the position of i, and other properties that will be explained below) over some alphabet Σ, with $@, \$ \not\in \Sigma$. Denote by Π the set of all possible properties. Then, we say that x is the *encoding* of some property π of node i iff $x = enc(i, \pi)$, where $enc : \mathbb{N} \times \Pi \to \Sigma$,

$$enc(i, \pi) = \begin{cases} \$e(i)\$ & \text{if } \pi = i, \\ \$e(i)@e(\pi)\$ & \text{otherwise.} \end{cases}$$

In other words, we have a standard encoding, except that each property of some node i (except i itself) is prefixed by an encoding of i. This will be useful when we put together the models of all the nodes that form an ad hoc network.

Each node i is characterized by its position, that changes with time. We denote by $p_i(t)$ the (encoding of the) position of node i at time t. In addition, each node has a set of characteristics that are invariant with time (for example, the transmission range). The structure of this set is, however, immaterial for the present discussion. Therefore, we consider that these characteristics are encoded by some string q_i for each node i. Finally, it is sometime assumed that each node has a constant velocity [13]. However, the constant velocity assumption is made for simulation purposes, and is not necessarily a feature of the real world. Indeed, the velocity of some node usually varies with time, and/or is unknown to the other nodes. Such a case is considered in [12], where the only thing known by any node is its current position. We consider here the most general case, where the only thing known about some node at some moment in time is its position at that moment.

Given a series of (conventional) words $w_1, w_2, \ldots$, we denote by $w_1 w_2$ the concatenation of w_1 and w_2. Moreover, we denote by $\sum_{i=1}^{\omega} w_i$ the (infinite) word obtained by successively concatenating the words w_i, $i \geq 1$

We are ready now to consider a timed ω-word that models some mobile host. A node i is modeled by the word $h_i = (\sigma, \tau)$, where $\sigma = (q_i)(\sum_{t=0}^{\omega} p_i(t))$, and $\tau = \tau_1 \tau_2 \ldots$, with $\tau_j = 0$ for $1 \leq j \leq |q_i p_i(0)|$, and, for any $k > 1$, $\tau_j = k$, $1 + |q_i| + \sum_{l=0}^{k-1} |p_i(l)| \leq j \leq |q_i| + \sum_{l=0}^{k} |p_i(l)|$.

In other words, the first part of h_i contains the invariant set of characteristics, together with the initial position of the object that is modeled. The time values associated with this subword are all 0. Then, the successive positions of the node are specified, labeled with their corresponding time values. It is immediate that all the necessary information about node i is contained in the word h_i.

7.3 A Model for Messages

Consider a message u issued at some time t. Such a message should contain the source node s and the destination node d. Furthermore, such a message may contain its type (for example, message or acknowledgment), the data that is to be transmitted, etc. All this content (referred to as the *body* of the message) is, however, immaterial, and we denote it by b_u as a whole. Considering that the encoding function e introduced above encodes messages over Σ as well, let the encoding of a message be $\$e(t)@e(s)@e(d)@e(b_m)\$$, and $k = |\$e(t)@e(s)@e(d)@e(b_u)\$|$. Then, the timed (finite) word that models u is $m_u = (\sigma, \tau)$, where $\sigma_1 \ldots \sigma_k = \$e(t)@e(s)@e(d)@e(b_u)\$$ and $\tau_j = t$ for $1 \leq j \leq k$.

Note that m_u is not a timed ω-word. On the other hand, it is easy to see that, for any node i, $h_i m_u$ is such a word. However, for a message to exist, there must be at least one node in the network, namely the node that sends it. That is, a model of a message would always be concatenated to the model of at least one node, and therefore the above construction is sufficient for our purposes.

Finally, one has to consider the model for the receiving event. For this purpose, assume that some message u (generated at time t_u, by a source s) is received by its intended destination d at some time t'_u. We model such an event by the timed word $r_u = (\sigma, \tau)$, where $\sigma_1 \ldots \sigma_{k'} = \$e(t)@e(s)@e(d)\$$ and $\tau_j = t'_u$ for $1 \leq j \leq k'$, with $k' = |\$e(t)@e(s)@e(d)\$|$. Again, such a word is not a timed ω-word, but the above argument still holds (namely, some "acknowledgment" cannot exists in a network with no hosts).

7.4 The Routing Problem

It is immediate that an ad hoc network with n nodes and without any message is modeled by the timed ω-word $a_n = h_1 h_2 \ldots h_n$. Then, a network of n nodes and some messages $u_1, u_2, \ldots, u_k$, $k \geq 1$, will be modeled by the word $w_{n,k} = h_1 h_2 \ldots h_n m_{u_1} m_{u_2} \ldots m_{u_k}$, and the model that includes the event of receiving u_i, $1 \leq i \leq k$ is $wr_{n,k} = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \ldots m_{u_k} r_{u_k}$. Moreover, given some countably infinite series of messages $u_1 u_2 \ldots$, the model of the network in which these messages are transmitted is $wr_{n,\omega} = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \ldots$. Note that $w_{n,\omega}$ is a timed ω-word under the reasonable assumption that any node can generate only a bounded number of messages per time unit.

In the following we may refer to the encoding m_u of a message u simply by "the message m_u". Whether the term message refers to a message or an encoding of a message will be clear from the context. For a fixed n, denote by N_n the set of all the words of the form $w_{n,k}$, $k \in \mathbb{N} \cup \{\omega\}$.

We are ready now to state the routing problem in ad hoc networks as a timed ω-language. Consider a network with n nodes, and a message u generated at time t, with body b, that is to be routed from its source s to the destination d. Then, a route of u is a word in the timed ω-language

$$R_{n,u} = \{w \in N_n\},$$

where, for some finite positive integer f, there exists a set of messages $u_1, u_2, \ldots, u_f$, and possibly a set of messages $rt_1, rt_2, \ldots, rt_g$, with g a positive, finite integer, such that $w = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} \ldots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \ldots m_{rt_f} r_{rt_f}$. Furthermore, for each message u_i, $1 \leq i \leq f$, denote by t_i, t'_i, s_i, d_i, and b_i the generation time, receiving time, source, destination, and body of u_i, respectively. Then,

1. $b_1 = b_2 = \ldots = b_f = b$, $s_1 = s$, $d_f = d$, $t_1 = t$,
2. for any i, $1 \leq i \leq f-1$, $d_i = s_{i+1}$, $t'_i = t_{i+1}$, and $range(s_i, d_i, t_i) = true$,
3. t'_f is finite.

In other words, the routing process generates f intermediate messages ($u_1, \ldots, u_f$). These are one-hop messages that contain the same information as the original message. Moreover, the time at which one such message arrives at the intended destination of u is finite (otherwise, the message is never received, and the routing process is hence unsuccessful). In addition, there might exist a finite number of additional messages ($rt_1, \ldots, rt_g$), that are exchanges between nodes in the routing process (for example, when the routing tables at each node are built/updated). In the following, we refer to some language $R_{n,u}$ as an (instance of a) *routing problem*, while some particular word $w \in R_{n,u}$ will be called an *instance* of $R_{n,u}$, or just routing instance when $R_{n,u}$ is understood from the context. Note that the actual routing (performed by some routing algorithm) of message u in some n-node network is modeled by a word in the corresponding routing problem.

Clearly, the language $R_{n,u}$ models all the relevant characteristics of a routing problem. Note that two routing algorithms may be compared by comparing their corresponding words from $R_{n,u}$. Moreover, more than one measure of performance may be considered. The measures of performance that are considered in [13] are the routing overhead (the total number of messages transmitted), path optimality (the difference between the number of hops a message took to reach its destination versus the length of the shortest possible path), and the message delivery ratio (the number of messages generated versus the number of packets received).

The first two measures have immediate correspondent in our model. Indeed, considering some word $w \in R_{n,u}$ corresponding to a routing algorithm, the routing overhead is given by $f+g$, the total number of messages that are generated. The number of hops a message traveled is given by $t'_f - t_1$. The message delivery ratio on the other hand needs some changes in our model, since we defined the routing problem as consisting in the successful deliveries of messages. Consider for this purpose the language $R'_{n,u} = \{w \in N_n\}$, where w has the same properties as above, except the finiteness of t'_f. This models a routing problem where the possibility of a message to be lost (that is, never received by its intended destination) exists. This property is modeled by the cases where $t'_f = \omega$.

However, note that in practice an infinite delivery time usually means that the delivery time exceeds some finite threshold T. This situation is modeled by our initial construction, where a lost message is a message for which $t'_f - t_1 > T$.

7.5 On Routing Algorithms

Up to now, we modeled the routing problem. Such an approach offers a basis for comparing routing algorithms, once the results of these algorithms are modeled as words from $R_{n,u}$. On the other hand, nothing is said about the routing algorithm itself. This happens in part because the notion of an acceptor for timed ω-languages is only sketched. However, one can outline the concept of a model for routing algorithms in ad hoc networks.

The immediate variant for such a model takes the form of an acceptor for the language $R_{n,u}$. However, further restrictions to such an acceptor must be imposed. Specifically, the real world router consists in n independent algorithms, that have limited means of communication. That is, two such nodes can communicate only by messages exchanged between them. A model for a routing algorithm must take this feature into account.

However, there is a second approach to this model, which is similar to the concept of parallel communicating grammar systems [20]. Such a system consists in a number of grammars, with their own work space, that communicate to each other by means of special symbols. Except for this communication, the grammars work independently. The case of parallel grammar systems closely resemble a real world ad hoc network[4]. Indeed, a node in such a network is unaware of the properties of another node, unless it receives a message from (or about) that node. Based on this intuition, we can propose a model for an n-node ad hoc network. For specificity, we model a routing instance $w = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} \ldots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \ldots m_{rt_f} r_{rt_f}$.

Such a model has n component timed ω-words H_i, $1 \le i \le n$, one for each node. Each H_i consists in a "local" component $\mathcal{L}_i$ and a "remote" component $\mathcal{R}_i$, where

$$\mathcal{L}_i = h_i m_{u_{j_1}} m_{u_{j_2}} \ldots m_{u_{j_x}} m_{rt_{k_1}} m_{rt_{k_2}} \ldots m_{rt_{k_y}}, \tag{12}$$

where $0 \le x \le f$, $0 \le y \le g$, $1 \le j_l \le f$ for any l, $1 \le l \le x$, and $1 \le k_l \le g$ for any l, $1 \le l \le y$. Moreover, the source of each message u_{j_l} or rt_{k_l} is i. That is, the local component consists only in those messages that are sent by the corresponding node, together with the space coordinates of that node.

Given $\mathcal{L}_i$, for each $j \ne i$, $1 \le j \le n$, denote by $M_{i,j}$ the set $\{r_{u_{j_l}} | 1 \le l \le x, d_{u_{j_l}} = j\} \cup \{r_{rt_{j_l}} | 1 \le l \le y, d_{rt_{j_l}} = j\}$. That is, the set $M_{i,j}$ contains the receiving events for all the messages that are sent from node i to node j. Then,

$$\mathcal{R}_i = v_1 \ldots v_k, \tag{13}$$

where $v_1 \ldots v_k$ are exactly all the elements in the set $\cup_{l=1}^{n} M_{l,i}$.

Finally, $H_i = \mathcal{L}_i \mathcal{R}_i$. In other words, the component H_i contains only those messages that are sent by the corresponding node, and those messages that are received by the node. Besides this information, no knowledge about the external world exists.

[4] It should be noted, however, that, while grammar systems are generative devices, the discussion here focuses on accepting devices instead. Therefore, the above parallel shall be taken exclusively as an intuitional support.

8 A Model for Distributed Computations

We presented in section 7.5 a distributed model for the routing algorithm. However, one can note that the above modeling of a routing problem can be extended to any distributed real-time algorithm.

Indeed, such an algorithm is composed of a set of processes, that execute independently, and communicate with each other exclusively by messages. Specifically, consider that there are n such processes. Consider now some process k isolated from the external world. It has to perform some real-time task, therefore, conforming to our thesis, its execution can be modeled by some timed ω-word. Call this word c_k. However, in addition to this computation, the process may send messages to other processes. Let these messages be modeled by some timed ω-word l_k. Note that the structure of l_k is similar to the structure of the $\mathcal{L}_k$ from section 7.5, relation (12), less the h_i part. Furthermore, the messages that are sent towards process k can be modeled by a timed ω-word r_k similar to $\mathcal{R}_i$ (relation (13)). Then, the behavior of process k is modeled by the word $c_k l_k r_k$, $1 \leq k \leq n$.

Therefore, such a model is not restricted to ad hoc networks, but is suitable for modeling any distributed real-time system instead. Whether this explicit representation of distributiveness is useful remains an open problem. We believe, however, that the above construction is natural and expressive. As noted above, a similar construction was studied in the context of conventional languages, namely the parallel communicating grammar systems (PCGS), introduced in [33] and further studied in, e.g., [14, 18, 20]. It is shown that the power of such devices is increased as compared to the power of usual grammars. We expect to find similar trends in the case of systems of ω-acceptors.

9 Conclusions, or Towards a Complexity Theory for 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.

Besides validating the thesis, the examples offered some interesting insights into the theory of real-time systems. Specifically, we constructed a recognition problem for queries in a real-time database system. While query complexity issues in traditional database systems were studied [2], the real-time domain received to our knowledge no attention. Nevertheless, the analysis of complexity of queries in this domain could be based on the newly developed recognition problem, which is yet another argument in favor of the mentioned complexity theory.

Furthermore, we presented a model for the routing problem in ad hoc networks. Not only did we formalize this problem, opening the road for a complexity analysis of it, but we also identified a variant of our model, suitable for modeling distributed real-time computations. Since there is a growing practical interest in distributed computations, such a model could be of interest. In particular, it offers an alternative to the real-time producer/consumer paradigm presented in section 3.2, that is not restricted to periodic message generation. Incidentally, note that the current developments in the area of wireless communications are tremendous, and this stresses the importance of theoretical analysis of routing algorithms in ad hoc networks, since such an analysis is not affected by the fast changing technological characteristics.

As a first step toward the suggested 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 [11], 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, the most important of these being time and 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 [4, 9, 15, 16, 17, 31], or can enhance significantly the quality of solutions [5, 6, 7, 8].

Note that a similar research was pursued in [34, 36], 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 [3]) is desirable.

References

[1] *Travelling exhibitions*, Vernissage: The Magazine of the National Gallery of Canada, 4 (Fall 1999), pp. 32–35.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.

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

[4] ——, *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.cs.queensu.ca/~akl/techreports/furniture.ps.

[5] ——, *Nonlinearity, maximization, and parallel real-time computation*, Tech. Rep. 99-433, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1999. http://www.cs.queensu.ca/~akl/techreports/realchaos.ps.

[6] 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.

[7] ——, *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.

[8] ——, *Real-time numerical computation: Beyond speedup III*, to appear in International Journal of Computers and their Applications, (March 2000). For a preliminary version see http://www.cs.queensu.ca/~akl/techreports/realnum.ps.

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

[10] 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.

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

[12] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward, *A distance routing effect algorithm for mobility (DREAM)*, in The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, 1998, pp. 76 – 84.

[13] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva, *A performance comparison of multi-hop wireless ad hoc network routing protocols*, in The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, 1998, pp. 85–97.

[14] S. D. Bruda, *On the computational complexity of context-free parallel communicating grammar systems*, in New Trends in Formal Languages, G. Păun and A. Salomaa, eds., Springer Lecture Notes in Computer Science 1218, 1997, pp. 256–266.

[15] 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.

[16] ——, *The characterization of data-accumulating algorithms*, Theory of Computing Systems, 33 (2000), pp. 85–96. For a preliminary version see http://www.cs.queensu.ca/˜bruda/www/data_accum2.

[17] ——, *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.

[18] L. Cai, *The computational complexity of linear PCGS*, Computer and AI, 15 (1996), pp. 199 – 210.

[19] J. Clifford and A. Crocker, *The Historical Relational Data model (HRDM) Revisited*, Benjamin/Cummings, CA, 1993, pp. 6–26.

[20] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, and G. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.

[21] 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.

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

[23] Z. J. Haas, *Panel report on ad hoc networks – Milcom'97*, Mobile Computing and Communications Review, 1 (1998), pp. 15–18.

[24] 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.

[25] S. Irani and A. R. Karlin, *Online computation*, in Approximation Algorithms for NP-Hard Problems, D. Hochbaum, ed., International Thomson Publishing, 1997, pp. 521–564.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] G. Özsoyoğlu and R. T. Sondgrass, *Temporal real-time databases: A survey*, IEEE Transactions on Knowledge and Data Engineering, 7 (1995), pp. 513–532.

[33] G. Păun and L. Sântean, *PCGS: The regular case*, Ann. Univ. Buc., Matem. Inform. Series, 38 (1989), pp. 55 – 63.

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

[35] 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.

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

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

[38] S. V. Vrbsky, *A data model for approximate query processing of real-time databases*, Data and Knowledge Engineering, 21 (1997), pp. 79–102.

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