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

Abstract

The concept of "real-time" has different meanings in the systems and theory communities. As a consequence, the currently available formal real-time models do not capture all the practically 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. Then, we use the developed formalism to model two important features of real-time algorithms, namely the presence of deadlines and the real-time arrival of input data. We also emphasize the expressive power of our model by using it to formalize aspects from the areas of real-time database systems and ad hoc networks. We offer formulations of the recognition problem for real-time database systems and the routing problem in ad hoc networks. Finally, we suggest a variant of our formalism that is suited for modeling parallel distributed real-time algorithms. We believe that the proposed formalism is a first step towards a unified and realistic complexity theory for real-time parallel computations.

Key words and phrases: Real-time computation, 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 (specifically, on reasoning about the correctness of programs) and on communication issues in distributed real-time systems.

By contrast, little work has been done in the direction of applied complexity theory, that is, the derivation of lower bounds and the design and analysis of algorithms for real-time computations. 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 [35] and further studied in [21, 30, 31]. Such a formalism offers many insights into the theory of real-time systems, but fails to capture many other aspects that are important in practice. Another model is the real-time producer/consumer paradigm, proposed in [23], 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 [10]. 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 [10] 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

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

imposed upon the computation, other applications require that input data are processed as soon as they become available, with more data to come while the computation is in progress. Variants (and combinations) of these two main requirements are often present. This complexity of the domain is probably the main obstacle towards a unified theory.

In this paper, we try to address this issue. We believe that the model of timed languages proposed in [10] 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 [10], but we apply such a definition to a larger extent, suggesting a general model for the acceptors of such languages.

Specifically, we introduce in section 3 a variant of the model from [10], called *well-behaved timed ω-languages*. Then, we present the notion of acceptance for such languages, together with the structure of an acceptor for them. We believe that our construction captures all the practical aspects of real-time computations. That is, our thesis is that *well-behaved timed ω-languages model exactly all real-time computations.*

In order to support this thesis, we show in section 4 how those ingredients that, when present, give to some problem the "real-time" qualifier (namely, computing with deadlines, and input data that arrive in real-time during the computation) can be modeled using our formalism. Then, the expressiveness of the formalism is emphasized in section 5, by modeling important practical problems. More precisely, we consider the domain of real-time database systems, and we offer a real-time variant of the *recognition problem* from the domain of classical database systems; then, we construct a formal model of the *routing problem* in ad hoc networks.

While the structure of a real-time algorithm as developed in this paper does not assume any particular machine architecture, an emphasis is given to parallel implementations. Indeed, we identify in section 6 a variant of our model, that is particularly suited for studying parallel and distributed real-time systems. This variant explicitly models the parallelism (and distributedness) of the system.

We believe that, starting from the formalism developed in this paper, a unified complexity theory for real-time systems can be naturally developed.

Section 2 reviews the notations used throughout the paper. Starting from this point, the paper can be viewed as split into two main parts: The first part (section 3) introduces the theory of timed ω-languages and real-time algorithms; section 3.2 concludes this first part. The second part (sections 4 and 5) presents real-time concepts and applications that are modeled using our formalism. The mentioned variant that is suitable for parallel and distributed systems is presented in section 6. We conclude the paper in section 7.

2 Preliminaries

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. It should be noted[1] that $\omega \notin \mathbb{N}$ [19]. 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 their concatenation. The length of a word σ is denoted by $|\sigma|$. $\mathbb{R}$ denotes the set of real numbers.

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 (denoted by $\sigma' \subseteq \sigma$) iff (a) for each σ'_i there exists a σ_j such that $\sigma'_i = \sigma_j$, and (b) for any positive integers i, j, k, l such that $\sigma'_i = \sigma_j$ and $\sigma'_k = \sigma_l$, it holds that $i > k$ iff $j > l$.

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.

[1] Also note that the cardinality of $\mathbb{N}$ is denoted by either ω [19] or $\aleph_0$ [32]. We chose the first variant in order to be consistent with the notation used in [10].

2.1 Timed automata

Since our models is based on the theory of timed automata, we review in what follows this theory. Our summary is based on [10].

The basis for the theory of timed automata is the ω-automaton. 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 [10] 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 with 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 concept 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 $i > 0$, there is a transition $(s_{i-1}, s_i, \sigma_i, l_i, d_i) \in \delta$ 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 is a *timed regular language*.

It should be noted that, even if a timed regular language looks well suited for modeling real-time computations, the TBA used in [10] for recognition of such languages is not sufficiently powerful for this purpose. We shall postpone this discussion till the next section.

3 Timed languages

While the notion of timed languages is very powerful, the device used for recognition of such languages in [10] (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 3.1 *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 3.2 *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 3.1. 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 3.1 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.

3.1 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 [10], but is not restricted to finite state acceptors.

However, our presentation is clearer if we use a slightly modified concept of time sequence:

Definition 3.1 A sequence $\tau \in \mathbb{N}^\omega$, $\tau = \tau_1\tau_2\ldots$, is a *time sequence* if it is an infinite sequence of positive values, such that the *monotonicity* constraint is satisfied: $\tau_i \leq \tau_{i+1}$ for all $i > 0$. In addition, a (finite or infinite) subsequence of a time sequence is also a time sequence.

A *well-behaved* time sequence is a time sequence $\tau = \tau_1\tau_2\ldots$ for which the *progress* condition also holds: for every $t \in \mathbb{N}$, there exists some finite $i \geq 1$ such that $\tau_i > t$. □

In should be noted that a time sequence may be finite or infinite, while a well-behaved time sequence is always infinite. In fact, a well-behaved time sequence in our terminology is identical to the concept of time sequence used in [10], except that, while time is considered dense in [10], we consider it to be discrete, since in essence the time perceived by a computer is discrete as well (besides, one can define a granularity of time as fine as desired).

Definition 3.2 A *timed ω-word* over an alphabet Σ is a pair (σ, τ), where τ is a time sequence, and, if $\tau \in \mathbb{N}^k$, then $\sigma \in \Sigma^k$, $k \in \mathbb{N} \cup \{\omega\}$. Given a symbol σ_i from σ, $i > 0$, then the associated element τ_i of the time sequence τ represents the time at which σ_i becomes available as input. A *well-behaved* timed ω-word is a timed ω-word (σ, τ), where τ is a well-behaved time sequence. A (well-behaved) timed ω-language over some alphabet Σ is a set of (well-behaved) timed ω-words over Σ. □

Definition 3.2 is an natural extension of the definition of timed regular languages presented in [10], except that we added the "well-behaved" qualifier, generated by the modified terminology presented in definition 3.1. We also take into consideration timed ω-words that are not well-behaved. Even if our thesis states that such words by themselves do not model real-time computations, they may be useful as intermediate tools in building real-time models.

3.1.1 Accepting timed languages

In light of the above definition, we can also establish the general form of an acceptor for timed languages:

Definition 3.3 A *real-time algorithm* A consists in a *finite control* (that is, a *program*), an *input tape* (that is, an *input stream*) that contains a timed ω-word, and an *output tape* (that is, an *output stream*) containing symbols from some alphabet Δ that are written by A. The input tape has the same semantics as a timed ω-word. That is, if (σ_i, τ_i) is an element of the input tape, then σ_i is available for A at precisely the time τ_i. During any time unit, A may add at most one symbol to the output tape. Furthermore, the output tape is *write-only*, that is, A cannot read any symbol previously written on the output tape. We denote by $o(A, w)$ the content of the output tape of some real-time algorithm A working on some input w. A may have access to an infinite amount of working storage space (working tape(s), RAM memory, etc.) outside the input and output tapes, but only a finite amount of this space can be used for any computation performed by the algorithm. □

It should be noted that the concept of working space has the same meaning as in classical complexity theory. Like a classical algorithm, a real-time algorithm can make use of some storage space in order to carry out the desired computation. When considering space-bounded real-time computations, we analogously consider the space used by the real-time algorithm as the amount of this storage space that is used during the computation, without counting the (content of) input and output tapes.

Let us fix some designated symbol $f \in \Delta$. The symbol f has the same meaning as the final state used in [10], where only those timed ω-languages accepted by finite automata are considered. However, since the configuration of a general machine may be hard to work with, we prefer to change the state with output. That is, a real-time algorithm accepts some word iff some designated symbol f appears infinitely many times on the output tape. Formally,

Definition 3.4 A real-time algorithm A *accepts* the timed ω-language L if, on any input w, $|o(A, w)|_f = \omega$ iff $w \in L$. □

It is worth mentioning that the actual meaning of the symbol f on the output tape might be different from algorithm to algorithm, but such a distinction is immaterial for the global theory of timed ω-languages. Indeed, consider an aperiodic real-time computation, e.g., a computation with some deadline. If, for some particular input, the computation meets its deadline, then, from now on, the real-time algorithm that accepts the language which models this problem may keep writing f on the output tape. That is, the first appearance of f signals a successful computation, and the subsequent occurrences of this symbol do not add any information, they being present for the sole purpose of respecting the acceptance condition (infinitely many occurrences of f). On the other hand, consider the timed language associated with a periodic computation, e.g., a periodic query in a real-time database system. Then, f might appear on the output tape each time an occurrence of the query is successfully served (obviously, a failure could prevent further occurrences of f, should the specification of the problem require that all the queries be served). In this case, each occurrence of f signals a successfully served query. However, even if the actual meaning of the f's on the output tape can vary from application to application, it is easy to see that the acceptance condition remains invariant throughout the domain of real-time computations.

It is assumed that the input of a real-time algorithm is always a (not necessarily well-formed) timed ω-word. That is, any real-time algorithm is fed with two sequences of symbols σ and τ, the first being a (possibly infinite) word over some alphabet, and the latter being the associated time sequence. It should be emphasized that a symbol σ_i with the associated time value τ_i is not available to the algorithm at any time t, $t < \tau_i$.

Note that a TBA is equipped with a set of clocks, since a finite automaton does not have access to any amount of storage space. However, a TBA has to keep track of time. Thus, clocks were provided as a convenient way of achieving this. One should also note that the restricted access a TBA has to its set of clocks prevents uses of this set in other ways than for time keeping. On the other hand, such a mechanism (that is, the set of clocks) is not mentioned in definition 3.3. This omission is intentional. As opposed to a TBA, a real-time algorithm has access to storage space, hence it can use (part of) this storage for time-keeping purposes.

The absence of clocks implies that, by contrast to TBA, there are no time constraints on state transitions in a real-time algorithm. Such constraints are made, however, immaterial by the semantics of the input tape. Indeed, definition 3.3 states that an input symbol is not available to the algorithm at a time smaller than the timestamp of that symbol. We believe that this constraint is sufficient since, conforming to definition 3.3, time restrictions are imposed by the input itself. One should note that real world real-time applications have the same property, namely that their behavior should conform to time restrictions imposed on their input and/or output rather than internal temporal constraints.

Should the need to define other classes of timed acceptors (where storage space is limited and/or the access to that storage space is restricted) arise, the set of clocks as a time keeping tool is likely to be needed again. For example, the definition of timed push-down automata can be obtained by naturally restricting definition 3.3, but one will have to add clocks to the model, given the limited (stack-like) nature of the storage space access of such a device. We believe that such models can be easily derived. However, the intended use of our model is building a complexity theory for general real-time computations, hence we will not consider such automata.

3.1.2 Operations on timed languages

Traditional formal language theory provides tools for generating new languages from existing ones, like union, concatenation, Kleene closure, etc. We therefore conclude the section that describes our model by defining equivalent operations on timed ω-languages.

The union, intersection, and complement for timed ω-languages are straightforwardly defined. Moreover, it is immediate that the language that results from such an operation on two (well-behaved) timed languages is a (well-behaved) 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 well-behaved timed 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. 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. Intuitively, such an operation is similar to merging two sequences of pairs (symbol, time value), that are sorted with respect to the time values. Formally, we have the following definition:

Definition 3.5 Given some alphabet Σ, let (σ', τ') and (σ'', τ'') be two timed ω-words over Σ. Then, 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. τ is a time sequence, that is, $\tau_i \leq \tau_{i+1}$ for any $i > 0$; both $(\sigma'_1, \tau'_1)(\sigma'_2, \tau'_2)\ldots$ and $(\sigma''_1, \tau''_1)(\sigma''_2, \tau''_2)\ldots$ are subsequences of $(\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$; furthermore, for any $i > 0$, there exists $j > 0$ and $d \in \{', ''\}$ such that $(\sigma_i, \tau_i) = (\sigma^d_j, \tau^d_j)$,

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, $i \leq k < l \leq j$, there exists m such that, for any $0 \leq \iota \leq j - i$, $(\sigma_{m+\iota}, \tau_{m+\iota}) = (\sigma^d_{i+\iota}, \tau^d_{i+\iota})$, and

3. for any positive integers i and j such that $\tau'_i = \tau''_j$, there exist integers 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 $L = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$. □

In addition to the mentioned order of the resulting sequence of symbols (formalized in item 1 of definition 3.5), two more constraints are imposed in definition 3.5. These constraints order the result in the absence of any ordering based on the arrival time, in order to eliminate the nondeterminism. First, if either

of the two ω-words contains some subword of symbols that arrive at the same time, then this subword is a subword of the result as well, and this is expressed by item 2. That is, the order of many symbols that arrive at the same time is preserved. Then, according to item 3, 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 3.6 Given some timed ω-language L, let $L^0 = \emptyset$, $L^1 = L$, and, for any fixed $k > 1$, $L^k = LL^{k-1}$. Furthermore, let $L^* = \cup_{0 \leq k < \omega} L^k$. We call L^* the Kleene closure of L. □

The following result is immediate:

Theorem 3.3 *The set of (well-behaved) timed ω-languages is closed under intersection, union, complement, concatenation, and Kleene closure, under a proper definition of the latter two operations. Furthermore, a subset of a (well-behaved) timed ω-language is a (well-behaved) timed ω-language.*

3.2 Comments

The term *real-time* is used by the complexity theorists in a somewhat different manner than in the real-time systems community. Indeed, the systems researchers use the term to refer to those computations in which the notion of correctness is linked to the notion of time [33]. By contrast, theorists often use real-time as synonym for *on-line* or *linear time* (the real-time Turing machine [31, 35] is probably the best example for the latter use). The model of well-behaved timed ω-languages intends to bridge this gap: While it is a formal model, it captures all the features of real-time computations as understood by the systems community. Therefore, this model have little in common to the theorists' real-time concept. However, we believe not only that well-behaved timed ω-languages accurately model the notion of real-time computation used in the systems community, but also that our model can be the basis of a meaningful complexity theory of real-time systems.

Claim 1 Well-behaved timed ω-languages model exactly all real-time computations.

We intend to investigate a complexity theoretic approach to real-time computations. In other words, our intended research direction is to define complexity classes for timed ω-languages, that capture an intuitive notion of real-time efficiency, and study the relations between these classes and between them and existing complexity classes. Classical complexity theory is of central concern not only for theorists, but also for practitioners. For example, the existence of a lower complexity bound for some problem is an important point: No matter how clever a program is, the bound cannot be overcome. Practitioners in the real-time systems area do not have such a theory to refer to. As a consequence, any question related to resource allocation and even solvability for a real-time problem is unique, in the sense that the answer to such a question should be developed from scratch (either by experiments or individualized proofs). A complexity theoretic approach to the real-time algorithms should offer a common ground to which practitioners can refer in order to get readily available answers to the mentioned questions.

Suppose that we prefix real-time complexity classes by "$rt-$." Then, some possible complexity classes might be $rt - \mathrm{SPACE}(f)$ (space-bounded real-time computations) and $rt - \mathrm{PROC}(f)$ (parallel real-time computations using a bounded number of processors), with their extensions $rt - \mathrm{PSPACE}$, $rt - \mathrm{LOGSPACE}$, $rt - \mathrm{LOGPROC}$, $rt - \mathrm{PPROC}$, etc. Then, the study or relation between these classes can be investigated. For example, an interesting question might be: Is the hierarchy $rt - \mathrm{PROC}(f)$ infinite? In other words, given any number k of processors, is there a well-behaved timed ω-language that can be accepted by a k-processor real-time algorithm but cannot be accepted by a $(k-1)$-processor one? If the answer is negative, then a practitioner looking at implementing some system might give less consideration to a parallel implementation, and focus first on some other options, such as obtaining a timely approximate solution instead of an exact solution that is available too late. On the contrary, should the answer be positive, considering a parallel implementation might become a high priority for our practitioner.

Finally, a note on the differences and similarities between timed ω-languages (that is, real-time algorithms) and classical formal languages (that is, classical algorithms) is in order. On one hand, it is immediate that formal languages are particular cases of timed ω-languages. Indeed, save for the time sequence, any word is a timed ω-word. If one relies on the semantics of the time sequence, one can add the time sequence $00\ldots0$ to a classical word and obtain the corresponding timed ω-word. However, none of the timed ω-words obtained in this manner is well-behaved. We have thus a crisp delimitation between real-time and classical algorithms, while keeping the formalisms as unified as possible.

4 Formal models for real-time concepts

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 order to further support this thesis, we take some meaningful examples, and we construct timed ω-languages that model them.

First, we model two general concepts that are central to real-time systems, namely *computing with deadlines*, and *real-time input arrival*. The appearance of either of these concepts in the specification of some problem gives the real-time characteristic to that problem [33]. Thus, being able to construct a suitable model for these concepts is crucial to the usefulness of the theory of timed ω-languages.

Once these concepts are successfully modeled, one can use those models in order to analyze practical real-time applications. Indeed, we shall take in section 5 two applications (namely, real-time database systems and routing in ad hoc networks), and use the results from this section in order to build formal models for them.

In general, given some problem, we denote the input and the output alphabets by Σ and Ω, 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, Ω, and $\mathbb{N}$ are disjoint. However, this does not reduce the generality of our constructions, since one can easily add some special delimiters in the proper places. Nonetheless, the presence of such delimiters will diminish the clarity of the constructions, hence we will omit them.

4.1 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 [24].

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 $\Sigma \cup \Omega \cup (\mathbb{N} \cap [max, 0]) \cup \{w, d\}$, $w, d \notin \Sigma \cup \Omega$ as follows:

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

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

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

(iii) This case is the same as case (ii), except that (2) becomes:

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise} \end{cases} \tag{3}$$

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 some "final" state s_f if they are identical, or some other designated state s_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 s_r on the whole acceptor. Otherwise, P_m compares the result computed by P_w with the proposed solution, and imposes either the state s_f or s_r, accordingly.

Once in one of the states s_f or s_r, the acceptor keeps cycling in the same state. In state s_f, the acceptor writes f on the output tape. The output tape is not modified in any other state.

It is immediate that the language accepted by the above acceptor is exactly $L(\Pi)$. It is also immediate that $L(\Pi)$ is well-behaved. Thus we completed the modeling of computations with deadlines in terms of ω-languages. Note that we assumed here that all the input data are available at the beginning of computation. However, the case when data arrive while the computation is in progress is easily modeled by modifying the timestamps that corresponds with each input data. But this case is covered in more details by our discussion in section 4.2.

4.2 Real-time input arrival

One of the computational paradigms that feature real-time input arrival is the *data accumulating paradigm*, that has been extensively studied in [14, 15, 26, 27]. The shape of input in this paradigm is very flexible, and any practically important form of real-time input arrival can be modeled by a particular class of problems within this paradigm. Therefore, we use in what follows the data-accumulating paradigm to illustrate the concept of real-time input arrival. Further models of this concept are also presented in section 5.

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:

$$f(n,t) = n + kn^{\gamma}t^{\beta} \tag{4}$$

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

Given a problem Π pertaining to this paradigm, we can build the corresponding timed ω-language $L(\Pi)$ similarly to section 4.1. 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: $\sigma_1 \dots \sigma_m = o$, $\sigma_{m+1} \dots \sigma_{m+n} = \iota_1 \dots \iota_n$, $\tau_i = 0$ 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 $\tau_{i_0+2i} = \tau_{i_0+2i+1} - 1$.

Now, an acceptor for $L(\Pi)$ has a structure which is identical[2] to the one used in section 4.1: It consists in the two processes P_w and P_m. P_w works exactly as the P_w from section 4.1, except that it emits some special signal to P_m each time it finishes the processing of one input data. Note that, since any d-algorithm is an on-line algorithm [15], 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 s_f or s_r is imposed upon the acceptor, accordingly).

Again, in state s_f, the acceptor writes f on the output tape, and the output tape is not modified in any other state. As well, once in one of the states s_f or s_r, the acceptor keeps cycling in the same state. It is immediate that $L(\Pi)$ is well-behaved and 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 [16, 26, 27] (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.

5 Applications

We modeled in sections 4.1 and 4.2 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 emphasize this aspect, we provide in what follows timed ω-languages that model problems from two highly practical areas, namely real-time databases (where we identify a real-time variant of the *recognition problem*) and ad hoc networks (where we offer a formal model for the *routing problem*).

5.1 Real-time database systems and timed languages

We start by reviewing in section 5.1.1 the main concepts of the real-time database systems theory. This review is mainly intended as a summary of the notations and concepts that are used later. Then, we enter the real-time domain, summarizing in section 5.1.2 the theory of real-time database systems. Finally, we use our formalism in section 5.1.3, where we model the recognition problem for real-time database systems.

5.1.1 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, $Dom(A) \subseteq$ **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

[2] 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.

Exhibitions

Title	*Description*	*Artist*
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

Schedules

City	*Title*	*Date*
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.

S

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

Figure 2: The result of a query.

be a relation or arity n. Then, a *tuple* over R is an expression $R(a_1, a_2, \ldots, a_n)$, 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.

In order to support the intuition behind the concepts presented above, let us consider as an example the relational database[3] 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, $Dom(Title) \supseteq \{\text{Terre Sauvage}, \text{Painter of the Soil}, \text{Sorrowful Images}\}$, 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.

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. For a

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

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:

$$\{enc(\mathbf{I})\$enc(u) | u \in q(\mathbf{I})\} \tag{5}$$

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].

5.1.2 Real-time database systems

The theory of real-time database systems (RTDBSs) may be viewed as the meeting point for the areas of active and temporal databases. In the following, we make a minimal presentation of this theory, directing the interested reader to [2, 28].

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:

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

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 5.1.2.

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, 28]. 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 is 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 (in general, 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, 28, 34].

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 [34], which derives from the historical relational data model [18].

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 and, 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 $d(x, y)$ 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 can be defined as a query language for real-time databases [18, 34].

Additional issues in the real-time database systems include the pattern of queries (periodic, sporadic, aperiodic), the nature of deadlines (hard, firm, soft) [28], 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 [34] 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 $(inst(S))^{\omega}$.

5.1.3 Real-time database systems as timed languages

As shown in section 5.1.1, 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, since such functions were widely used (see for example [2, 25]). 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 5.1.2. 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[4] $q = |enc(o_k)|$; 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, $1 \leq k \leq r$. 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:

[4] 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.

$$db_B = db_0 db_1 \dots db_r \tag{6}$$

We have now a model for real-time databases, which it trivially a well-behaved timed ω-language. 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 [28].

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 4.1, 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 4.1 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:

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

(iii) This case is the same as case (ii), except that (7) becomes:

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise} \end{cases} \tag{8}$$

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 well-behaved, since the concatenation of an infinite number of well-behaved timed ω-words is not necessarily well-behaved. In our case, however, the result of the concatenation is a well-behaved timed ω-word, and this follows immediately from the following observation.

Lemma 5.1 *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 5.1 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 (well-behaved) timed ω-language:

$$L_{aq} = \{db_B aq_{[q,s,t]} | s \in q(B)\} \tag{9}$$

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 (well-behaved) timed ω-language:

$$L_{pq} = \{db_B pq_{[q,s,t,t_p]} | s \in q(B)\} \tag{10}$$

□

Note that the recognition problem for real-time queries is similar to the same problem for conventional queries, shown in (5), except that the (conventional) words used in (5) are replaced by timed ω-words.

5.2 Ad hoc networks

We direct our attention now to another real-time problem, namely the routing problem in ad hoc networks. We show how to model this problem using the theory of timed ω-languages. In the process, we also identify an interesting variant of real-time algorithms, which we believe to be useful in modeling parallel distributed real-time systems.

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 [12, 22]. 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, [11, 12] 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 [12], 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 [16]. 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.

5.2.1 Assumptions and notations

When speaking about ad hoc networks, we 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.

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.

5.2.2 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 $@, \$ \notin \Sigma$. 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 [12]. 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 [11], 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 $\prod_{i=1}^{\omega} w_i$ 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 $\sigma = (q_i)(\prod_{t=0}^{\omega} p_i(t))$, and $\tau = \tau_1 \tau_2 \ldots$, with $\tau_j = 0$ for $1 \leq j \leq |q_i p_i(0)|$, and, for any $k > 1$, $\tau_j = k$, $1 + |q_i| + \prod_{l=0}^{k-1} |p_i(l)| \leq j \leq |q_i| + \prod_{l=0}^{k} |p_i(l)|$.

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.

5.2.3 A model for messages

Now that we have a model for the set of nodes, all we have to do is to connect them together. We have that is to model message exchanges between nodes.

Consider a message u issued at some time t. Such a message should contain the source node s and the destination node d. In addition, 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 well-behaved 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 $\sigma_1 \dots \sigma_{k'} = \$e(t)@e(s)@e(d)\$$ 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 well-behaved, but the above argument still holds (namely, some "acknowledgment" cannot exists in a network with no hosts).

5.2.4 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 $a_n = h_1 h_2 \dots h_n$. Then, a network of n nodes and some messages $u_1, u_2, \dots, u_k$, $k \geq 1$, will be modeled by the word $w_{n,k} = h_1 h_2 \dots h_n m_{u_1} m_{u_2} \dots m_{u_k}$, and the model that includes the event of receiving u_i, $1 \leq i \leq k$ is $wr_{n,k} = h_1 h_2 \dots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \dots m_{u_k} r_{u_k}$. Moreover, given some countably infinite series of messages $u_1 u_2 \dots$, the model of the network in which these messages are transmitted is $wr_{n,\omega} = h_1 h_2 \dots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \dots$. Note that $w_{n,\omega}$ is a well-behaved 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 $R_{n,u} \subseteq N_n$, where, for some finite positive integer f, there exists a set of messages $u_1, u_2, \dots, u_f$, and possibly a set of messages $rt_1, rt_2, \dots, rt_g$, with g a positive, finite integer, such that any word $w \in R_{n,u}$ has the form $w = h_1 h_2 \dots h_n m_{u_1} r_{u_1} \dots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \dots m_{rt_f} r_{rt_f}$. 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. $b_1 = b_2 = \dots = b_f = b$, $s_1 = s$, $d_f = d$, $t_1 = t$,
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, ..., 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, ..., 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 [12] 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 $R'_{n,u} \subseteq N_n$, where any word $w \in R'_{n,u}$ 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 $t'_f - t_1 > T$.

5.2.5 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. The immediate variant for such a model takes the form of real-time algorithm that accepts the language $R_{n,u}$. However, further restrictions to such an acceptor must be imposed: 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: 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 $w = h_1 h_2 \dots h_n m_{u_1} r_{u_1} \dots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \dots m_{rt_f} r_{rt_f}$.

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

$$\mathcal{L}_i = h_i m_{u_{j_1}} m_{u_{j_2}} \dots m_{u_{j_x}} m_{rt_{k_1}} m_{rt_{k_2}} \dots m_{rt_{k_y}} \tag{11}$$

where $0 \leq x \leq f$, $0 \leq y \leq g$, $1 \leq j_l \leq f$ for any l, $1 \leq l \leq x$, and $1 \leq k_l \leq g$ for any l, $1 \leq l \leq y$. 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 $j \neq i$, $1 \leq j \leq n$, denote by $M_{i,j}$ the set $\{r_{u_{j_l}} | 1 \leq l \leq x, d_{u_{j_l}} = j\} \cup \{r_{rt_{j_l}} | 1 \leq l \leq y, d_{rt_{j_l}} = j\}$. 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:

$$\mathcal{R}_i = v_1 \dots v_k \tag{12}$$

where $v_1 \dots v_k$ 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.

6 Explicit parallel and distributed models

We presented in section 5.2.5 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.

The parallelism is introduced in the theory of timed languages by the structure of acceptors for such languages. Indeed, an implementation for such an acceptor can be considered for any underlying model of computation. We implied such a feature when mentioning multiprocessor implementations and the corresponding real-time complexity classes. However, a first disadvantage of such an approach is that the real-time complexity classes that involve the notion of parallelism are likely to be different from model to model (in particular, the class $rt - \text{PROC}(f)$ might be different when the underlying model of computation is changed from, say, the PRAM to an interconnection network). Then, no information about the explicit distributed

character of the computation is present in the language itself. Since our goal is to present a consistent theory of real-time computations, it is desirable to have such an information right within the language.

One can note that a similar construction was studied in the context of conventional languages, namely the *parallel communicating grammar systems* (PCGS), introduced in [29] and further studied in, e.g., [13, 17, 20]. A PCGS 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[5]. Based on this intuition, we can propose a model for parallel real-time computations: Consider a parallel algorithm with n processing elements. In general (that is, without any assumption about the parallel machine on which this algorithm is implemented), one can assume that the implementation is composed of a set of n processes, that execute independently, and communicate with each other by messages. Consider now some process k isolated from the external world. It has to perform some real-time task, therefore, 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 all these messages be modeled by some timed ω-word l_k. Furthermore, the messages that are received by process k can be modeled by a timed ω-word r_k. Then, the behavior of process k is modeled by the timed ω-word $c_k l_k r_k$, $1 \leq k \leq n$.

If some real-time algorithm consists in p such processes, then it's behavior is modeled by the tuple $(c_1 l_1 r_1, \ldots, c_p l_p r_p)$.

In particular, the real-time algorithm presented in section 5.2.5 is just a particular case of this construction. Furthermore, the PRAM can be considered a particular case as well: Since the communication between different processors is accomplished by read/write operations from/to the shared memory, there is no communication. That is, both l_k and r_k are null words in a system of timed ω-acceptors implemented on the PRAM.

We believe that, once the theory for timed ω-languages is in place, the study of the distributed model presented above is worthwhile.

7 Conclusions, or towards a complexity theory for real-time computations

We believe that the notion of timed languages and real-time algorithms as introduced in section 3 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. 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.

We also 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 [23], 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.

We believe that this paper offers the basis (as well as the the motivation) for the development of a complexity theory for real-time systems. In general, such a theory takes into account the measurable resources

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

used by an algorithm, the most important of these being time and space. In the real-time environment, however, 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 for real-time computations with respect to the number of available processors is a very interesting direction, with promising prospects. (Recall here that it has been already established that a parallel approach can make the difference between success and failure [4, 9, 14, 15, 16, 27], or can enhance significantly the quality of solutions [5, 6, 7, 8] in real-time environments.)

We note in closing that a similar research was pursued in [30, 31], 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. The theory of well-behaved timed ω-languages offers a foundation for this pursuit.

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*, in Proceedings of the Conference on Parallel and Distributed Computing Systems, Cambridge, MA, November 1999, pp. 227–233.

[5] ——, *Nonlinearity, maximization, and parallel real-time computation*, in Proceedings of the 12th Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, November 2000.

[6] S. G. Akl and S. D. Bruda, *Parallel real-time optimization: Beyond speedup*, Parallel Processing Letters, 9 (1999), pp. 499–509. For a preliminary version see http:// www.cs.queensu.ca/ ~akl/ techreports/ beyond.ps.

[7] ——, *Parallel real-time cryptography: Beyond speedup II*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June 2000, pp. 1283–1290. For a preliminary version see http:// www.cs.queensu.ca/ ~akl/ techreports/ realcrypto.ps.

[8] ——, *Parallel real-time numerical computation: Beyond speedup III*, International Journal of Computers and their Applications, 7 (2000), pp. 31–38. 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] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science, 126 (1994), pp. 183–235.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] ——, *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.

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

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

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

[19] J. H. Conway, *On Numbers and Games*, Academic Press, 1976.

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

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

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

[32] S. Swierczkowski, *Sets and Numbers*, Routledge & Kegan Paul, London, UK, 1972.

[33] USENET, *Comp.realtime: Frequently asked questions*, Version 3.4 (May 1998). http:// www.faqs.org/ faqs/ realtime-computing/ faq/.

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

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