Distributed Timed Automata

Padmanabhan Krishnan

Department of Computer Science
University of Canterbury, PBag 4800
Christchurch, New Zealand
E-mail: paddy@cosc.canterbury.ac.nz

Abstract

In this article we discuss (i) a model suitable for describing a distributed real-time system and (ii) a notion of implementation for such systems on a uniprocessor system. The first point is addressed by amalgamating finite state automata with dense time (or Alur-Dill automata) and asynchronous distributed (or Zielonka) automata over a distributed alphabet. The second point is addressed by defining the process of interleaving and time slicing.

1 Introduction

A physically distributed system consists of a number of processors interconnected in some fashion. A program for such a system is developed as a collection of potentially communicating processes. During the implementation phase each process is assigned to some particular processor. To simplify matters, one may assume that each process is essentially sequential. Otherwise, one can decompose a non-sequential process into a set of communicating processes. A simple way to study such systems is to represent communication as synchronisation. Such a model leads to a tractable analysis as seen in languages such as occam [4]. Theoretically, trace models are appealing as discussed in [6] and [10]. In this model, each process is represented as a finite automaton having its own alphabet. Processes are forced to synchronise on common symbols. For example, all the processes that have a in their alphabet need to participate by individually exhibiting an a. Only then can an a be exhibited by the global system. Two symbols are said to be independent if they have no process in common. Independence is an indication of what may occur "simultaneously" in a truly distributed or parallel environment. For example, if process 1 and 2 can exhibit a and process 3 and 4 can exhibit b, then a and b are independent. Processes 3 and 4 are not tied up while the global system exhibits an a. If processes 5 and 6 can both exhibit actions c and d, the actions c and d are not independent. Process 5, being sequential, cannot exhibit both c and d

in the same step. Independence can be used to identify essentially equivalent behaviours thereby obtaining a partial order model.

The above discussion pertains mainly to untimed systems. We now discuss existing work related to uniprocessor timed systems. Finite state timed automata [2] are a natural extension of finite automata to timed behaviour where the underlying notion of time is continuous. The timing constraints involve only rational constants. This restriction provides decidability for certain essential properties such as determining emptiness. The article [2] describes a model for parallel timed systems. However, it is very restrictive. The parallel composition of two timed traces is either empty or a single time trace. This is because a single common time frame is assumed. The only relaxation from the standard model is that independent actions can occur at the same time. In other words, parallel composition corresponds to intersection.

In a distributed system each processor has its own notion of time. Although the various clocks can be kept in synchronisation, there is no single global clock that is used by all processors. To presuppose the existence of identically synchronised distributed clocks is too severe. The behaviour of timed programs (and hence timed processes) could depend on the allocation of the processes to processors. Fixing a single time frame does not support the discussion of issues such as clock synchronisation, distributed consensus etc.

For the purposes of modular construction, it is useful to have a separate notion of time for each processor. Each module can be treated as a separate process and can assume to have its own time frame. The composition of modules will retain these separate time frames. Behaviour at remote sites will not necessarily affect the behaviour at the local module. The presence of individual times is also useful in the development of timed independence, as this results in the separation of causal and temporal independence. Such a semantics can be easily incorporated into formal description techniques like message sequence charts [5,8] which have a natural notion of distribution. This will become clearer when it is discussed in more detail in section 3.

The presence of multiple time threads raises another issue. One must also have ways of combining different time frames to create a uniform single notion of time. This would be useful in implementing parallel machines on other machines with fewer processors. This is common in the area of real-time systems where one specifies it as a collection of independent/semi-independent tasks. The next step is to actually implement this specification, say for the sake of simplicity, on a uniprocessor system.

In our setting, we show that we require an extension of the standard timed automata to discuss the notion of implementing distributed automata. We show that this extension does not violate the key property of decidability of language emptiness.

The main contribution of this paper is the development of a natural notion of distributed timed automata. We present a definition of implementation along with the generalisation of timed automata that is needed. Results to justify the various definitions are also presented.

There are two pieces of related work. [7] introduces a partial order model for timed systems. However it is based on the existence of an independence relation on the transition relation. [3] also introduces a partial order model. But the definition is based on a product of processes construction. There is no discussion of implementation in both these cases.

In the next section we present a quick review of distributed automata and timed automata. The rest of the paper is concerned with the development of distributed timed automata and the notion of implementation. In this paper we restrict ourselves to finite behaviours. It is not too hard to extend these results to infinite behaviours.

2 Notation and Prior Work

Asynchronous Automata

First we present a quick overview of asynchronous automata (or untimed distributed automata). To simplify matters no distinction between processes and processors is made. We assume a finite set of process or agent names, say Loc identified as $\{1, 2, \ldots n\}$. These identify the individual processes in the system. Associated with each process l belonging to Loc is a finite automaton $\mathcal{A}_l$. Therefore, there is a finite alphabet Σ_l and a finite set of states Q_l which we refer to as a local alphabet and state of agent l. We let Σ denote the union of all the various alphabets. For every a in Σ, we define $loc(a)$ to be the set of agent names which have a in their alphabet. That is, $loc(a) = \{i \mid a \in \Sigma_i\}$. The idea is that all the agents in $loc(a)$ have to co-operate (i.e., synchronise) to exhibit a. The structure $(\Sigma_1, \Sigma_2, \ldots \Sigma_n)$ is referred to as the distributed alphabet while the structure $(\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n)$ is referred to as the collection of local automata.

Two actions a and b are independent if $loc(a)$ is distinct from $loc(b)$, i.e., $loc(a) \cap loc(b) = \emptyset$. Independent actions do not have any agent in common. Hence, if possible, the independent actions can be exhibited "simultaneously".

Towards modelling behaviour, we define $Q = \bigcup_{i \in Loc} Q_i$ to be the set of all possible local states. Given $L \subseteq Loc$, we define the set of possible L-states, $Q_L = \{ s\colon L \longrightarrow Q \mid \forall\, l \in L,\ s(l) \in Q_l \}$. For a given $q \in Q_L$ and $S \subseteq L$ we let q_S and $q(S)$ denote the restriction of q to S. That is, $q_S = q\mid_S$ viz., given a sub-system we can identify the states of the components of the sub-system. For every action a in Σ we write Q_a to represent $Q_{loc(a)}$. The transitions are defined for each action. That is, the set of a-transitions $\Rightarrow_a$ is a set of pairs of a-states. That is, $\Rightarrow_a \subseteq Q_a \times Q_a$.

The overall behaviour of the system (or the behaviour of the distributed automaton) is defined as follows. For $q, q' \in Q_{Loc}$, $q \xrightarrow{a} q'$ if and only if (1) $(q_{loc(a)}, q'_{loc(a)}) \in \Rightarrow_a$ and (2) $q_{Loc-loc(a)} = q'_{Loc-loc(a)}$.

The global (distributed) transition relation requires the participation of all the relevant agents to exhibit an action. All the other agents make no move. Let $Q_{in} \subseteq Q_{Loc}$ denote the set of initial states and $F \subseteq Q_{Loc}$ denote the set of final states. As for finite automata, a word w is accepted if and only if there is a run (i.e., sequence of transitions) of the global automaton over w starting in some initial state and terminating in a final state.

A key result is that if two actions are independent (i.e., the set of processes needed for the two actions is disjoint), they can be performed in either order. This is the classical diamond property.

Proposition 1 ([10]) If $q \xrightarrow{a} q_1$ and $q_1 \xrightarrow{b} q_2$ such that $loc(a) \cap loc(b) = \emptyset$, there exists q_3 such that $q \xrightarrow{b} q_3$ and $q_3 \xrightarrow{a} q_2$.

Consider a distributed automaton $\mathcal{A}$ with independent actions a and b. The string $w_1 a b w_2$ is accepted if and only if the string $w_1 b a w_2$ is also accepted.

Timed Automata

The following is a quick review of timed automata as presented in [2] but restricted to finite behaviours. A timed automaton is a finite automaton equipped with a set of timers (or clocks) C. These timers are used to measure the time elapsed since a particular transition. Hence a timed automata is a structure $(Q, \Sigma, C, \longrightarrow, Q_{in}, F)$ where Q is a finite set of states, Σ the input alphabet, C the set of clocks, $Q_{in} \subseteq Q$ the set of initial states and $F \subseteq Q$ the set of accepting states. Towards the definition of the transition relation $\longrightarrow$ assume a set of timing constraints $(\phi(C))$ expressed over C as follows:

$$\beta ::= x \leq c \mid c \leq x \mid \neg\beta \mid \beta_1 \wedge \beta_2$$

where x is a timer in C and c a rational constant. One can assume a set of primitive constraints where a timer is compared against a rational constant. The set of timing constraints is then defined to be the set of finite boolean combinations of primitive constraints. The transition relation of the timed automaton is annotated with a timing constraint and a set of clocks that have to be reset. That is, a timed transition relation $\longrightarrow$ is a subset of $Q \times \Sigma \times$

$\phi(C) \times 2^C \times Q$. We often write $q \xrightarrow[X]{a,\delta} q'$ to indicate that (q, a, δ, X, q') belongs to the relation.

The words (say of length n) accepted by such an automaton are pairs of maps (σ, τ) where $\sigma : \{0..n-1\} \longrightarrow \Sigma$ and $\tau : \{0..n-1\} \longrightarrow \mathbb{R}^{\geq 0}$. That is, it consists of a sequence over the alphabet along with timing information for each element in the sequence. A run over such a word is a map $\rho : \{0..n-1\} \longrightarrow Q \times V$ where V is the set of *clock interpretations* $\{v \mid v : C \longrightarrow \mathbb{R}^{\geq 0}\}$ such that the following conditions hold. We write (q_i, v_i) for $\rho(i)$.

- $\rho(0) = (q_0, v_0)$ where q_0 is the initial state and $v_0(c) = 0$ for every clock c.
- For every $i \geq 0$ there exist β and X such that
  · $(q_i, \sigma_i, \beta, X, q_{i+1})$ is a valid transition
  · $v_i + \tau(i) - \tau(i-1)$ satisfies β
  · $v_{i+1}(x) = 0$ if $x \in X$ otherwise $v_{i+1}(x) = v_i(x) + \tau(i) - \tau(i-1)$

A run identifies the duration that elapses between two actions. The clocks are updated appropriately and the updated clocks are used to determine if the timing constraint is satisfied. If the transition is taken the relevant clocks are reset to 0. Such a run is accepting if the state component of $\rho(n-1)$ is a final state. That is, a computation terminating in a final state is accepted (the clock values at the end are of no concern.)

Proposition 2 ([2]) *The emptiness question is decidable for timed automata. That is, given a timed automaton, the question whether it accepts any word is decidable.*

Various algorithms to decide various properties (such as emptiness) depend on what is called the region construction. That is, an untimed automaton whose states represent the actual state along with equivalent clock interpretations is constructed. The regions constructed depend on all the clock constraints present in the automaton. We will appeal to this concept in our paper. The reader is referred to [9] for an excellent tutorial on issues related to timed automata.

3 Time and Implementation

Thus far we have reviewed existing work. This section presents the main contribution of this work. The first issue is regarding time. We would like to obtain a commutativity like result (similar to proposition 1) for distributed timed automata. But a single global time frame restricts the type of results that can be obtained. If a and b are independent it is not always the case that a timed sequence involving a and b can be commuted. For example, $(a, 4)(b, 5)$ could be a valid observation generated by the transitions $q_1 \xrightarrow{a, X>3} q_1'$

and $q_2 \xrightarrow{b,Y>4} q_2'$. However the observation $(b,5)(a,4)$ is not a valid observation as it is ill timed. One can reconcile this by considering causality and differentiating between timing and causality [1]. Hence by adding more causality information we could deem $(b,5)(a,4)$ as a well timed trace. Here we adopt a simpler solution. We make it explicit that the time frames are different. In the untimed case when a particular action is output, nothing happens at the agents not involved in the action. We use this interpretation but reinforce the meaning of nothing happens to indicate that time remains static. Therefore, time is not represented by a single real value, but by a tuple of values. That is we are using the well known idea of vector time for distributed systems.

The number of components to a global time could depend on the independence structure of the actions. For example, if agents 1 and 2 have identical alphabets, one could use a single time value to represent time at both the agents. However, this is complicated as common time could introduce dependencies not present in the distributed alphabet. Also, a single time frame assumes that both agents proceed at the same clock rate. This need not be the case as slow machines could still synchronise with fast machines. Hence to simplify matters and also handle realistic conditions, each agent will have its own notion of time.

Having introduced a multiple notion of time we would still like to define a notion of implementation. The implementation could have fewer processes than the original specification. Therefore we need some way of mapping the times from the specification to the implementation. When we compile a collection of processes to run on a uniprocessor environment some form of interleaving occurs. Hence the addition of elapsed time is natural for an interleaving semantics. We would like these definitions to be suitable for both traces and the automata that generate them. We first introduce distributed timed automata and then discuss their implementation.

3.1 Distributed Timed Automata

As in the case of untimed automata, we have to choose what is globally available. In our model, clocks (or timers), like the input symbols, can be made global. At first we describe a very general (with respect to the visibility of timers) model to illustrate the various issues. But later we restrict this to a simpler and more realistic model.

As before let $Loc = \{1 \dots n\}$ be the set of agents with each agent (say l) having its own alphabet Σ_l and state space Q_l. Let C be the set of globally available clocks or timers. For every clock c belonging to C, we let $tloc(c)$ denote the agent that controls (updates) the clock. That is, we are assuming that every timer is resident on exactly one of the set of available processors. We also let $tloc$ be applied to a set in which case $tloc(S)$ denotes the set $\{tloc(c) \mid c \in S\}$.

Locations/Agents	$\{1, 2 \cdots n\}$
Distributed Alphabet	$(\Sigma_1, \Sigma_2, \ldots \Sigma_n)$
Local Automata	$(\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n)$
Global Set of Clocks	C
Agent controlling clock x	$tloc(x)$
Clocks visible to action a	$ts(a)$
Local transitions	$\Rightarrow_a$
Global Transition (or System Behaviour)	$\longrightarrow$

Fig. 1. Distributed timed automata

For each a in Σ we let $ts(a)$ denote the set of clocks accessible (or visible) to a. This is the global information used in the transition system. Thus any action can read/reset clocks but there is a designated processor that updates each clock. Note that the above definition allows an action to use remote clocks (i.e., clocks not necessarily at its locations). More precisely, for any action a there is no restriction on the relationship between $tloc(ts(a))$ and $loc(a)$. It is this condition that will be strengthened later.

For each input symbol a we associate a transition system satisfying the following condition. If $q \overset{\delta}{\underset{X}{\Longrightarrow}}_a q'$, it is required that X is a subset of $ts(a)$ and δ is a timing constraint over $ts(a)$. As for asynchronous automata we will have a set of initial (say Q_{in}) and final states (say Q_F) each of which is a subset of Q_{Loc}. Figure 1 summarises the various constituents of a distributed timed automaton.

Before we define the system behaviour (denoted by $\longrightarrow$) (or the joint behaviour of the individual local automata) of the distributed automaton, we need a few auxiliary definitions. These are generalisations of the definitions used for timed automata [2].

At any given instant the *global time* is a vector, i.e., a collection of the time at each agent. We let gts denote the set of all possible global times. That is, $gts = \{f \mid f : Loc \longrightarrow \mathbb{R}^{\geq 0}\}$.

The notion of *clock interpretations* is as before. That is, each clock holds only a single real value. Clock interpretations are defined to satisfy timing constraints in the usual fashion. Therefore, semantically a timing constraint is a boolean combination of (perhaps different) local constraints.

Given t an element of gts and ν a clock interpretation we define the *t successor of ν* as $\nu^t(c) = \nu(c) + t(tloc(c))$. That is, the clock c is advanced by the appropriate component of t. For t_1 and t_2 belonging to gts, we let $t_1 - t_2$ represent the time obtained by subtracting the values at each component.

Given global states $q, q' \in Q_{Loc}$, an input symbol $a \in \Sigma$, clock interpretations ν, ν' and a global time t we say $(q, \nu) \xrightarrow[t]{a} (q', \nu')$ iff

(1) $\forall l \notin loc(a)$: $q(l) = q'(l)$ and $t(l) = 0$ while $\forall l \in loc(a)$: $t(l) > 0$
(2) There exists a transition $q_{loc(a)} \xRightarrow[X\ a]{\delta} q'_{loc(a)}$ where ν^t satisfies δ and for every x in X $\nu'(x) = 0$ and for every y not in X $\nu'(y) = \nu^t(y)$.

The key observation in the above definition is that time elapses only when an action is exhibited. Hence while exhibiting the action a the time at all the agents not involved in the action a does not change. The clocks at the agents not involved in the action a move may be reset to 0 only if they are visible to a.

The following example, illustrates this behaviour. Let $loc(a) = \{1, 2, 3\}$. Let the clocks c_1 on agent 3, c_2 on agent 4 and c_3 on agent 5 be visible to a. Formally, $tloc(c_1) = 3$, $tloc(c_2) = 4$, $tloc(c_3) = 5$ and $ts(a) = \{c_1, c_2, c_3\}$. On agent 1, the execution of a updates time at locations 1, 2 and 3. Time is unchanged at locations 4 and 5. Only clock c_1 will get an updated value. This can be viewed as agent 3 sending the updated time along with the synchronisation message to agent 1. The current values of c_2 and c_3 (which will not be changed as agents 4 and 5 do not participate to exhibit a) will be used to determine if the timing constraints are satisfied.

The above behaviour is based on the assumption that each agent receives time updates of all the relevant remote agents. Such an assumption is essential as we have not placed any restriction on $tloc(ts(a))$ with respect to $loc(a)$.

Definition 3 A timed word is a finite sequence $(a_0, t_0)(a_1, t_1)(a_2, t_2) \cdots (a_{k-1}, t_{k-1})$ where $\forall 0 \leq i < k-1, \forall j \in Loc$ $t_i(j) \leq t_{i+1}(j)$ and $t_i \neq t_{i+1}$ This word is *accepted* if there is a sequence of transitions $(q_i, \nu_i) \xrightarrow[t_i - t_{i-1}]{a_i} (q_{i+1}, \nu_{i+1})$ for i belonging to $\{1 .. k-1\}$ where (i) $q_0 \in Q_{in}$, (ii) $q_k \in Q_F$ (iii) $\nu_0(c) = 0$ for all $c \in C$ and (iv) $t_{-1} = 0$

Let $\mathcal{L}(\mathcal{A})$ represent the language accepted by the automaton $\mathcal{A}$.

We now define a notion of independence such that a form of commutativity result holds. Note that it is not possible for both $(a, t_1)(b, t_2)$ and $(b, t_2)(a, t_1)$ to be accepted as time cannot decrease. In exhibiting b some component of t_1 has to be increased to obtain t_2. However, if $(a, t_1)(b, t_2)$ is part of a valid behaviour, it might be possible for one to obtain a new time stamp t_3 such that $(b, t_3)(a, t_2)$ is also valid.

If one can commute a and b, it is clear that the components of time that a uses or alters must be independent of the components used or altered by b. As every untimed automaton is essentially a special type of a timed automaton, it is clear that the intersection of $loc(a)$ and $loc(b)$ should be empty. The corresponding condition on the set of clocks is not sufficient. That is, demanding only $ts(a) \cap ts(b) = \emptyset$ does not suffice. Let $loc(a) = \{1\}$, $ts(a) = \{y\}$, $tloc(y) = 2$, $loc(b) = \{2\}$, $ts(b) = \{x\}$ and $tloc(x) = 1$. Now the behaviour $(a, t_1)(b, t_2)$ may not always yield $(b, t_2)(a, t_1)$ as when b is exhibited the value of y changes and hence (a, t_1) may no longer be viable.

Hence we have to consider the locations of all the various clocks that an action may use. Let Dt_a be $tloc(ts(a))$ and Dt_b be $tloc(ts(b))$. Requiring that Dt_a and Dt_b be disjoint is sufficient but not necessary. Let $loc(a) = \{1\}$, $ts(a) = \{x, z\}$, $tloc(x) = 1$, $tloc(z) = 3$, $loc(b) = \{2\}$, $ts(b) = \{y, u\}$, $tloc(y) = 2$ and $tloc(u) = 3$. Now, $Dt_a = \{1, 3\}$, $Dt_b = \{2, 3\}$ and their intersection is not empty. While exhibiting a or b the time at location 3 is static. So the values of z or u will not increase. Of course, they can reset to 0. But resetting z (u) can be done by a (b) and is of no concern to b (a). Therefore, a and b are really independent. The appropriate definition of independence for timed automata is precisely stated below.

Definition 4 Two actions a and b are independent iff

(1) $loc(a) \cap loc(b) = \emptyset$ and $ts(a) \cap ts(b) = \emptyset$
(2) $\forall x \in ts(a), tloc(x) \notin loc(b)$ and $\forall y \in ts(b), tloc(y) \notin loc(a)$

Independence means both causal independence and temporal independence. Point 2 indicates that "remote" clocks can be used provided they are associated with "neutral" agents. That is, any timer used by a should not be controlled by an agent that can exhibit b and vice-versa. These are required as transitions involving these actions can reset the value of the remote clocks.

Proposition 5 Let $\mathcal{A}$ be a distributed timed automaton with independent actions a and b. Then, $(q, \nu)\xrightarrow[t_a]{a}(q', \nu')\xrightarrow[t_b]{b}(q'', \nu'')$ implies the existence of q_b and ν_b such that $(q, \nu)\xrightarrow[t_b]{b}(q_b, \nu_b)\xrightarrow[t_a]{a}(q'', \nu'')$.

Proof: The proof for the state part of the result follows directly from the definition of the global transition relation and the definition of independence. One needs to verify that the timing aspects are satisfied.

As $(q, \nu)\xrightarrow[t_a]{a}(q', \nu')\xrightarrow[t_b]{b}(q'', \nu'')$, there are transitions of the form $q\xrightarrow[X]{a,\delta_a}q'$ and $q'\xrightarrow[Y]{b,\delta_b}q''$.

As a and b are independent it is easy to see that the transitions $q\xrightarrow[Y]{b,\delta_b}q_b$ and $q_b\xrightarrow[X]{b,\delta_a}q''$. are possible. We now have to show that $\nu + t_a$ satisfies δ_a iff $\nu_b + t_a$ satisfies δ_a and $\nu + t_b$ satisfies δ_b iff $\nu' + t_b$ satisfies δ_b. The following observations ensure this result.

For every c in X, $\nu'(c) = 0$ and as $tloc(c)$ does not belong to $loc(b)$, $\nu''(c) = 0$. As such a c does not belong to $ts(b)$, $\nu_b(c)$ is identical to $\nu(c)$.

For every c that does not belong to X but where $tloc(c)$ belongs to $loc(a)$, $\nu'(c)$ has value $\nu(c) + t_a(tloc(c))$ which is identical to $\nu''(c)$.

For every c that does not belong to X and $tloc(c)$ that does not belong to $loc(a)$, $\nu'(c)$, $\nu(c)$ and $\nu''(c)$ are all identical.

Hence the satisfaction of the timing constraints follow.

□

Proposition 6 Let L be a language accepted by a distributed timed automaton with independent actions a and b. Then, $w(a,t)(b,t')w'$ belongs to L iff for some t'', the word $w(b,t'')(a,t')w'$ belongs to L.

Proof: The result follows directly from proposition 5. □

3.2 Restricted Model

For the purposes of discussing an implementation, we consider a slightly simpler model. We require that the location of every timer used by an action a is one of the locations of a. Exhibiting an action can therefore change the time only at those locations participating in the transition. Hence an action cannot influence timers at arbitrary locations. This can be stated formally as: $\forall x \in ts(a), tloc(x) \in loc(a)$.

Note that for a given clock assignment to locations and availability of clocks to actions, the general model is *strictly more powerful* than the simpler model. Let clock x be assigned to location 1 and clock y assigned to location 2. Let actions a and b be allowed to use clocks x and y. In the general model let a belong only to agent 1 and b belong only to agent 2. Consider the following transitions $q_1 \xrightarrow[x]{a,x<1\wedge y=0} q_1'$ and $q_2 \xrightarrow[y]{b,x=0\wedge y<1} q_2'$. If (q_1', q_2') is a final state the word $(a, < 0.5, 0 >)(b, < 0.5.0.5 >)$ is accepted. That is, when the action a is exhibited the time at location 2 does not change and similarly when the action b is exhibited the time at location 1 does not change. However, in the simpler restricted model, a and b would need to belong to both agent 1 and 2. Hence time cannot stay static at any location when the actions are exhibited. This is stated precisely in proposition 7

Proposition 7 For some C, $tloc$ and ts and ${\Sigma_i}^s$ for each i belonging to Loc, there exists a language L that is accepted by a generalised distributed timed automaton but no restricted automaton with $\Sigma_i \supseteq {\Sigma_i}^s$ accepts L.

Proof: Let $C = \{x, y\}$ such that $tloc(x) = 1$ and $tloc(y) = 2$. Let $ts(a) = \{x, y\}$ and $ts(b) = \{y\}$. Let $loc(a) \supseteq \{1\}$ and $loc(b) \supseteq \{2\}$. Let L consist of only $(b, < 0, 1.0 >)(a, < 1.0, 1.0 >)$. It is easy to see that the following general automaton with start state (q_1, q_2) and final state (q_1', q_2') accepts this language.

$$q_1 \xrightarrow[\emptyset]{a, x=1 \wedge y=1} q_1' \qquad q_2 \xrightarrow[\emptyset]{b, y \leq 1} q_2'.$$

Now to the restricted case. As b occurs when the time at agent 1 is 0, b cannot belong to the alphabet of agent 1. Similarly, as the time at location 2 is static when a is exhibited, a cannot belong to the alphabet of agent 2. Hence no distributed alphabet can satisfy the requirement imposed by the clock distribution and accept the above language. □

Note that the above result is valid for a given distribution of clocks. This is similar to the distinction between product and asynchronous automata over a given distributed alphabet. One can always construct an automaton to accept that language if one is allowed to change the syntactic structure.

We presented the general model only to indicate the various possibilities in defining a distributed timed system. While one can describe a notion of implementation for such system, it is unnecessarily complex. The implementation technique described below for the restricted model can be adapted to the general case.

3.3 Implementation Issues: Interleaving Automata

When implementing a distributed system on a uniprocessor system, interleaving of various actions is essential. When considering systems with timing constraints one would ideally like to implement them without altering the timing constraints. That is, syntactically we would not like to change the timing constraints. Semantically we recognise the existence of multiple local times. Consider a system where the action a occurs after 5 units of time and an action b (independent from a) occurs within 3 units of time. In the distributed setting one can perform the a before the b. But that should not rule out b. While implementing this one should recognise the difference in the notions of time used to verify the constraint associated with a and b. The time on the uniprocessor system must be ‘sliced’ appropriately to the subsystems containing a and b. One may wish to avoid such ‘sliced’ allocations. In that case, at least sometimes, it is possible to alter the timing constraints syntactically. On interleaving a followed by b, the new constraint on b would require it to wait for 3 time units after a. This would be specified by using a new timer that is reset on an a. However such a syntactic definition is hard to formulate in a consistent and compositional fashion.

In order to describe such interleaved systems nicely, one needs a model that is slightly more powerful than the standard Alur-Dill [2] model. The extra power is gained by the ability to hold (i.e., not incrementing) the value of various timers while exhibiting an action. The intuition is that such timers belong to a different site and hence the scheduling of another process (due to interleaving) should not update their times.

To identify the clocks which are incremented at the same rate we assume an equivalence relation on the set of clocks called *sloc*. The interpretation of (c_1, c_2) belonging to *sloc* is that both c_1 and c_2 are from the same process. Therefore, at every step both c_1 and c_2 will be incremented by the same amount. This will always hold except when one of the clocks is reset explicitly. Intuitively, we associate an agent with each equivalence class and let $sloc_i$ denote the class associated with agent i. Hence there will be n non-empty equivalence classes (i.e., one for each agent in the original system).

Given *sloc* and a subset of the clocks in system (say X) we define $UE(X) = \{y \mid x \in X \text{ and } sloc(x, y)\}$. The set $UE(X)$ identifies all the clocks that are (update) equivalent to some clock in X.

Note that for every action a, all the clocks in $ts(a)$ need not belong to the same equivalence class under *sloc*. That is because a could belong to the alphabet of various agents. Hence the different clocks in $ts(a)$ could be updated at different rates.

Given a global vector time we add up the individual components to obtain a single real value as the time at the interleaved automaton. Notationally, given $t \in gts$, we let $st(t)$ denote the sum of the various time components. That is, $st(t) = \sum\limits_{i \in Loc} t(i)$.

In the interleaved automaton an action a will be exhibited only when all the original agents have excecuted their transitions to exhibit a. Hence the various delays are added to form a single delay.

The structure of an interleaving automaton is identical to that of an Alur-Dill automaton. The transitions are of the form $q \xrightarrow[X]{a,\delta} q'$ where we require that all the clocks mentioned in δ and X belong to $ts(a)$. Hence the implementation of a distributed automaton does not change the basic transition structure. It is the notion of a timed step (and hence run and acceptance) that is different. Formally, an implementation or interleaving automaton is represented by (Q, Σ, $\longrightarrow$, Q_0, F, ts,, $sloc_1$, $\cdots$ $sloc_n$). The number of equivalence relations indicates the number of agents in the original automaton. Figure 2 summarises the structure.

The behaviour of an interleaving automaton (Q, Σ, $\longrightarrow$, Q_0, F, ts, $sloc_1$, $\cdots$ $sloc_n$) is described below. As before we define a timed step (presented in Definition 8). Based on this we have the usual notion of acceptance, i.e., the

Agent/Automaton	Only one
Clock Usage	From original description
Clock Equivalence Classes	Number of agents in the distributed system
Alphabet	Single alphabet structure
Transition (or System Behaviour)	A single relation

Fig. 2. Interleaving automaton

existence of a run that ends in a final state.

Definition 8 Let ν be any clock interpretation. A time step $(q, \nu) \xrightarrow[t]{a} (q', \nu')$ is defined iff the following conditions hold.

(1) $q \xrightarrow[X]{a,\delta} q'$
(2) $\exists t_1, t_2, \ldots t_n$ such that
  (2a) $\sum_{j \in \{1..n\}} t_j = t$
  (2b) $\nu^t \models \delta$ where ν^t is defined as: $\nu^t(c) = \nu(c) + t_i$ where $t_i \neq 0$ iff $(c \in UE(ts(a)) \wedge c \in sloc_i)$.
  (2c) $\nu'(c) = 0$ if $c \in X$ otherwise $\nu'(c) = \nu^t(c)$.

For a given ν and t if conditions 2a and 2b hold we say that ν^t is a time successor to ν under a. This is almost identical to the notion in [2]. The difference is that it is dependent on the action. Now by the usual region construction but using the above as the definition of time successor, the following result holds.

Proposition 9 *The emptiness question is decidable for interleaving automata.*

Proof: The main principle of the proof is still based on the region construction. The principal difference is in the construction of the untimed automaton where the notion of the new time successor is useful. The key observation is that the clocks are either updated or remain unchanged. This implies that the region information is sufficient. □

We define the following for notational convenience. In a distributed automaton with a set of clocks C we let C_i denote the set of clocks controlled by agent i. That is, $C_i = \{c \mid tloc(c) = i\}$. The following proposition states the fact that every distributed timed automaton can be translated to an equivalent interleaving automaton.

Proposition 10 Given a distributed timed automaton over the structure (Σ_1 $\ldots \Sigma_n$), and (C_1, $\ldots C_n$), accepting the language L, there is an interleaving automaton with clock equivalence classes $sloc_1$, $\ldots sloc_n$ derived from (C_1, $\ldots C_n$) accepting L_I such that

$(a_0, t_0)(a_1, t_1) \cdots (a_n, t_n) \in L$ iff $(a_0, st(t_0))(a_1, st(t_1)) \cdots (a_n, st(t_n)) \in L_I$.

Proof: The result can be shown by translating a run in the original automaton to a run in the interleaving automaton. □

Properties

We now consider other results concerning interleaving automata. Various results concerning asynchronous automata are valid over a given distributed alphabet. Similarly, the results for interleaving automata are valid for given clock equivalences. As non-determinism is permitted, it is easy to see that interleaving automata are closed under union maintaining the clock structure.

Proposition 11 Let $\mathcal{A}_1$ be $(Q_1, \Sigma, \longrightarrow_1, Q_0^1, F_1, sloc_1^1, \cdots sloc_n^1)$ and accept the language L_1. Similarly, let $\mathcal{A}_2$ be $(Q_2, \Sigma, \longrightarrow_2, Q_0^2, F_2, sloc_1^2, \cdots sloc_m^2)$ and accept the language L_2. There is an automaton that accepts $L_1 \cup L_2$ such that if timers x and y belong to same (or different) equivalence classes in the constituent automata, they belong to same (or different) classes respectively in the union automaton.

Proof: We will assume the following conditions hold.

$Q_1 \cap Q_2 = \emptyset$. For every i, j $sloc_i^1 \cap sloc_j^2 = \emptyset$ and
Without loss of generality, $n \leq m$.

It is easy to construct an automaton $\mathcal{A} = (Q, \Sigma, \longrightarrow, Q_0, F, sloc_1, \cdots sloc_m)$ where

$Q = Q_1 \cup Q_2$, $Q_0 = Q_0^1 \cup Q_0^2$, $F = F_1 \cup F_2$,
$\longrightarrow = \longrightarrow_1 \cup \longrightarrow_2$,
For i such that $1 \leq i \leq n$, $sloc_i = sloc_i^1 \cup sloc_i^2$ and for j such that $n < j \leq m$ $sloc_j = sloc_j^2$.

It is easy to show that this automaton accepts the union of the two languages.
□

In the above definition, we have not insisted on disjoint clocks. If two structures use the same (named) clock, we assume that there is only one clock in the joint structure.

We now discuss the intersection operation. It is easy to show that interleaving automata are not closed under intersection if one has to maintain clock equivalences. That is, if clocks in separate equivalence classes cannot be identified, one cannot, in general, obtain the intersection of two interleaving automata. This appears to be a serious drawback. Without such a result, automata theoretic techniques to model checking etc. would be rendered useless. We now argue that this is not really such a major issue.

There are two main problems why interleaving automata are not closed under intersection. The first is that the division of global time may not be consistent across machines. The second is that such a division occurs only once. This means that some clocks have to be shared. This is to force the updating of a clock in one automaton to automatically update the clock for the other automaton. However, when clocks are shared, resetting of the clocks causes difficulty. If $q_1 \xrightarrow[X_1]{a,\delta_1} q_1'$ and $q_2 \xrightarrow[X_2]{a,\delta_2} q_2'$ are transitions in the two automata, neither the union nor the intersection of X_1 and X_2 is correct for the intersection automaton. Hence to handle the resetting of clocks in a clean fashion, one has to assume that the set of clocks are disjoint.

We present an example to illustrate the time division issues. Consider the two automata (each having only one transition) $q_1 \xrightarrow[\emptyset]{a,\delta_1} q_1'$ and $q_2 \xrightarrow[\emptyset]{a,\delta_2} q_2'$ where $\delta_1 = (2 < x < 3) \wedge (7 < y < 8)$ and $\delta_2 = (u = 5) \wedge (v = 5)$. For the first automaton let $sloc_1^1$ contain x and $sloc_2^1$ contain y and for the second automaton let $sloc_1^2$ contain u and $sloc_2^2$ contain v. If q_1' and q_2' are the accepting states, the timed string $(a, 10)$ is accepted by both the automata. For the intersection if we create four distinct equivalence classes, the string $(a, 10)$ cannot be accepted as 10 cannot be broken into four parts and yet satisfy the individual timing constraints. Merging the various equivalence classes also does not help as the break up of the 10 units of time is quite different for the two automata. Hence it is difficult to construct the intersection automata and yet retain the original timing constraints.

What is essentially required is an identical equivalence and update structure but disjoint individual clocks. For that it is useful to examine the original automaton which yielded the interleaving automaton.

The first automaton is the interleaved implementation of the parallel automaton. $q_{11} \xrightarrow[\emptyset]{a,\delta_{11}} q_{11}'$ and $q_{21} \xrightarrow[\emptyset]{a,\delta_{21}} q_{21}'$ where $\delta_{11} = (2 < x < 3)$ and $\delta_{21} = (7 < y < 8)$. The clock x belongs to the first agent and y belongs to the second agent. The second automaton can also be suitably decomposed to yield $q_{12} \xrightarrow[\emptyset]{a,u=5} q_{12}'$ $\quad q_{22} \xrightarrow[\emptyset]{a,v=5} q_{22}'$.

Now it is easy to see that the two parallel automata do not have any word in common. The second automaton can only accept the word $(a, <5, 5>)$ which cannot be accepted by the first automaton. Therefore, it is not surprising that the 'intersection' of the implementations does not accept anything. Hence interleaving automata are 'closed' under a notion of intersection with respect to the 'original' automata. This is defined below.

Let $\mathcal{A}_1$ use clocks C_1 and have n-clock equivalence classes (called $sloc_1^1$, $sloc_2^1$, $\ldots sloc_n^1$). Let $\mathcal{A}_2$ be similar and use clocks C_2 and have n-clock equivalence classes (called $sloc^2_1$, $sloc^2_2$, $\dots sloc^2_n$). We will assume that C_1 and C_2 are disjoint. We also require that for every a in the alphabet, if there is a c such that $c \in ts^1(a)$ and $c \in sloc^1_i$ then there is a c' such that $c' \in ts^2(a)$ and $c' \in sloc^2_i$. This will ensure that the right clocks are updated in the intersection machine. For the intersection automaton define $sloc_i = sloc^1_i \cup sloc^2_i$ and $ts(a) = ts^1(a) \cup ts^2(a)$. For every transition $q_1 \xrightarrow[X_1]{a,\delta_1} q'_1$ in $\mathcal{A}_1$ and $q_2 \xrightarrow[X_2]{a,\delta_2} q'_2$ in $\mathcal{A}_2$ add $(q_1, q_2) \xrightarrow[X_1 \cup X_2]{a,\delta_1 \wedge \delta_2} (q'_1, q'_2)$.

The following proposition states that the construction of the intersection automaton for interleaving automata actually corresponds to the intersection of the distributed timed automata that gave rise to the interleaving automata.

Proposition 12 *If A over $(\Sigma_1 \dots\ \Sigma_n)$ and $(C_1 \dots\ C_n)$ is implemented by $\mathcal{A}_1$ over $(C_1 \dots\ C_n)$ and if B over $(\Sigma_1 \dots\ \Sigma_n)$ and $(C'_1 \dots\ C'_n)$ is implemented by $\mathcal{A}_2$ over $(C'_1 \dots\ C'_n)$ the intersection of A and B is implemented by the intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Expressive Power

We conclude this discussion with a small example. This is a timed version of the asynchronous automaton used in [10] to show that asynchronous automata are strictly more powerful than product automata. Let clock x belong to agent 1 and clock y belong to agent 2. Let Σ_1 be $\{a,\ c\}$ and Σ_2 be $\{b,\ c\}$. The transition system for the two agents is given below.

$$\begin{tikzcd}
p_1 \arrow[r, "{a,\delta_x}"] & p_2 \arrow[r, "{a,x>2}"] \arrow[l, bend left, "{c,\delta,R}"] & p_3 \arrow[ll, bend right, "{c,true,R}"'] & & q_1 \arrow[r, "{b,\delta_y}"] & q_2 \arrow[r, "{b,y>2}"] \arrow[l, bend left, "{c,\delta,R}"] & q_3 \arrow[ll, bend right, "{c,true,R}"']
\end{tikzcd}$$

where δ_x is $1 < x < 2$, δ_y is $1 < y < 2$, δ is $x < 2 \wedge y < 2$ and R is the set $\{x, y\}$.

The implementation automaton is the usual product automaton but with no c transitions from the states (p_2, q_3) and (p_3, q_2). When an a is exhibited the clock y is not updated and when b is exhibited the clock x is not updated. Hence it is easy to see that the region $(1 < x < 2) \wedge (1 < y < 2)$ is a b successor to $(1 < x < 2) \wedge (y = 0)$.

An example of a time word is $(a, t_0)(b, t_1)(b, t_2)(a, t_3)(c, t_4)$ where $t_2 - t_0 > 2$ and $t_3 - t_2 + t_0 > 2$. Note that the delay between the second b and the second a is not specified explicitly. The delay is essentially governed by the timing constraint on the second a which depends on when the first a was exhibited.

This example is now used to show that implementation automata are strictly more powerful than timed automata.

Proposition 13 There is no timed automaton that accepts the same language as the above implementation automaton.

Proof: The proof is based on the operations that can be performed on the various clocks and the set of permissible timing constraints. The following partial transition structure is essential to accept the required language with C the set of available clocks.

$$\begin{tikzcd} r_0 \arrow[r, "{a,1<x<2}", "Z_0"'] & r_1 \arrow[r, "{b,1<y<2}", "Z_1"'] & r_2 \arrow[r, "{a,\delta}", "Z_2"'] & r_3 \end{tikzcd}$$

The task now is to populate the various set of clocks that get reset with timers. That is, we have to specify the contents of the sets Z_0, Z_1 and Z_2. We also have to identify a suitable timing constraint on the a transition from r_2. Clearly y should belong to Z_0. Any clock (generically denoted by x_0) that does not belong to either Z_0 or Z_1 is measuring time since the start of the computation. Any clock (generically denoted by x_1) that belongs to Z_0 but not Z_1 has the time since the a was exhibited. Any clock (generically denoted by x_2) that belongs to Z_1 has the time that has elapsed since the action b was exhibited. The timing constraint δ should reflect the fact that the global time minus the time used by the action b should be greater than 2. This can be expressed only as $x_2 > 2 - (x_0 - x_1)$. This timing constraint is clearly illegal as it involves clock arithmetic. Following the result in [2] this cannot be expressed as a legal constraint. Hence a timed automaton that accepts the language cannot exist. $\square$

The above result shows that the notion of time regularity has to be modified if it has to be used in the context of distributed systems.

It is easy to check that every interleaving automaton can be modeled as an Alur-Dill automata if addition in clock constraints is permitted. However, in general, emptiness for automata with addition over clock variables is not decidable. Emptiness is decidable for the interleaving automaton as the additions over clock variables occur in a very constrained fashion.

4 Conclusion

In this paper we have developed a theory of timed distributed (asynchronous) automata. The main feature is the presence of multiple time frames. This allowed us to obtain a natural notion of independence and commutativity. We presented two varieties of the model. We showed that one is strictly weaker than the other. The more general model imposed no restriction on the relationship between the alphabet structure and the clock structure. In the restricted model, the alphabet structure and the clock structure had to be consistent. This was followed by the definition of implementation for the restricted model. The definition of implementation required a model of timed automata that was more general than the standard Alur-Dill automata. However, the important question of language emptiness was still decidable.

References

[1] L. Aceto and D. Murphy. On the Ill-timed but Well-Caused. In E. Best, editor, *CONCUR 93*, volume LNCS-715, pages 97–111. Springer Verlag, 1993.

[2] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98*, volume LNCS-1466, pages 485–500. Springer Verlag, 1998.

[4] INMOS Ltd. **occam-2** *Reference Manual.* Prentice Hall, 1988.

[5] S. Mauw and M. A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37, 1994.

[6] A. Mazurkiewicz. Basic Notions of Trace Theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume LNCS 354, pages 285–363. Springer Verlag, 1989.

[7] F. Pagani. Partial Orders and Verification of Real-Time Systems. In B. Jonsson and J. Parrow, editors, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume LNCS 1135, pages 327–346, Uppsala, Sweden, 1996. Springer Verlag.

[8] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Basic Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12), June 1996.

[9] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, volume LNCS-1494, pages 114–152, Veldhoven, The Netherlands, 1998. Springer Verlag.

[10] W. Zielonka. Notes on Finite Asynchronous Automata. *RAIRO: Theoretical Informatics and Applications*, 21(2):101–135, 1987.