Sādhanā, Vol. 17, Part 1, March 1992, pp. 1–28. © Printed in India.

Modelling real-time systems: Issues and challenges

R K SHYAMASUNDAR and S RAMESH*

Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India
*Department of Computer Science and Engineering, Indian Institute of Technology, Powai, Bombay 400 076, India

Abstract. In this paper, we discuss the issues and challenges that lie in the specification, development, and verification of real-time systems. In our presentation, we emphasize on the issues underlying modelling of real-time distributed concurrency.

Keywords. Real-time; reactive systems; concurrency; bisimulation; trace equivalence; scheduling.

1. Introduction

Real-time systems are designed to cater to many applications ranging from home appliances or laboratory instruments to process control systems, flexible manufacturing, flight control and tactical control in military applications. Flexible manufacturing is a special kind of real-time application where the behaviour of each manufacturing machine can be adapted *instantaneously* to continuously changing working conditions while still satisfying a global optimality criterion. In flight control systems *real-time* automatic manoeuvering is used to achieve significant reduction of fuel consumption and also for tactical control over the target. In these systems, the timely execution of requests and responses by the computers is critical to the successful operation of both the physical systems and the computer itself. That is, in addition to the normal functional requirements, it is necessary that responses to inputs (from the environment) must happen in a given interval of time. We refer to these systems as real-time systems and the specified intervals of time as *deadlines*. We use the qualification *reactive* to refer to the fact that the system has to respond to environment stimuli continuously. In such systems one can distinguish two kinds of deadlines.

- *Hard deadlines*: Here, it is important that the deadline must be met; otherwise the result is useless; in other words, *what is needed is the right output at the right time.*
- *Soft deadlines*: In these deadlines, not meeting the deadlines results in the degradation of the system performance.

One of the common concepts that counter a majority of the process control systems is that of providing continual feedback to an unintelligent environment. The continual demands of an unintelligent environment cause these systems to have relatively rigid and urgent performance requirements, such as real-time response requirements and *fail-safe* reliability requirements. It seems that this emphasis on performance

requirements is what really characterizes time-critical systems, and causes us to be more aware of their roles in their environments than we are for other types of systems. The interface between a process control system and its environment tends to be complex, asynchronous, highly parallel and distributed. This is another direct result of the *process control* concept, because the environment is likely to consist of a number of objects which interact with the system and each other asynchronously in a parallel fashion. Furthermore, it is probably the complexity of the environment that necessitates computer support in the first place. This characteristic makes the requirements difficult to specify in a way that is both precise and comprehensible. Finally, embedded systems can be extraordinarily hard to test. The complexity of the system/environment interface is one obstacle, and the fact that these programs often cannot be tested in their operational environments is another. It is not feasible to test flight-guidance software by flying with it, nor to test ballistic-missile-defence software under battle conditions. Further, embedded systems are especially likely to have stringent resource requirements. These are requirements on the resources, mainly physical in this case, from which the system is constructed. This is because embedded systems are often installed in places (such as satellites) where the weight, volume or power consumption must be limited, or where temperature, humidity, pressure and other factors cannot be as carefully controlled as in the traditional machine room. It is important to note that a failure quite often results in economic, human and ecological catastrophes. Thus, safety and reliability are extremely important for time-critical process control systems. Various parameters one has to cope up with in building such systems can be seen from some of the main characteristics of real-time systems given below.

(a) The system tends to be large, complex and can be extraordinarily hard to test.
(b) The environment that the system interacts with is nondeterministic. That is, most of the times, there is no way to anticipate in advance the precise order of external events.
(c) High speed external events (perhaps in parallel), must be able to affect the flow of control in the system easily.
(d) The requests must be responded and handled within certain bounded time limits.
(e) The system is a coordinated set of asynchronous distributed units.
(f) The mission time is long. The system not only must deal with ordinary situations but also must be able to recover from some extraordinary ones.

It must be quite evident from the above characteristics that the design of complex real-time systems poses a serious challenge since many of the requirements and restrictions are often conflicting with one another. Thus, one of the most important needs is to design sound methodologies for the specification, verification and development of real-time systems that would support the common requirements of flexibility and predictability of systems. This would certainly go a long way in bridging the thin line between acceptable and unacceptable systems.

In this paper, we discuss the issues and challenges that lie in the specification, development, and verification of real-time systems with an emphasis on the modelling of distributed real-time concurrency. The rest of the paper is organized as follows: § 2 discusses aspects of real-time systems that make it different from other systems and the notion of time; § 3 surveys the issues of modelling real-time reactive systems in some detail as the study provides a basis for observation-based specifications. The challenges in the design of real-time systems are highlighted in § 4 followed by a discussion in § 5.

2. Characteristics of real-time systems

In this section, we discuss the need of explicit time, the difference between real-time and traditional systems and the problem of real-time system design.

2.1 *What is the purpose of explicit notion of time?*

Traditional programs describe transformations that change values of variables in discrete steps. Any processor implementing these transformations takes a finite amount of time. In the interest of generality, programs are usually designed such that the computed results are independent of the execution speed of their processor(s). In other words, *time* considerations are completely irrelevant for the functional behavour of programs and their correctness; perhaps it is only relevant for questions of schedule and efficiency.

To avoid the need to cope with explicit time considerations even in the case of concurrent programming, a common agreement has been evolved to use the concept of nondeterminism to abstract from concrete time to handle classes of processes working with different relative speeds. Such an approach helps to avoid harmful comparisons of execution times and thus, provides highly abstract semantic models for non-sequential programs. The only indispensable assumption we need is that the processor have non-zero finite speed. Adherence to execution-time independence affords the tremendous advantage that a program's validity can be deduced solely from the static program text containing logical assertions on the state of the computations after each statement and signal exchange. If we depart from this rule and let our program's validity depend on the execution speed of the utilized processors, we enter the area commonly called *real-time* programming (Wirth 1977). There are two main reasons for designing time-dependent programs.

(i) One of the principal reasons for consideration of execution-time dependent programs in the case of concurrent programming systems is that certain processes are not programmable at discretion, as they may be part of the environment; this leads to situations wherein processes fail to wait for synchronization signals indicating completeness of the cooperating partner's task. As a result, cooperation with such processes will necessarily have to depend on processor speed.
(ii) The other important reason for considering time explicitly is the case of reactive systems that model some physical process; here, the internal laws which define the *natural* behaviour of the physical process are functions of a parameter referred to as *physical time.* The need for reference to absolute time (or clock) for these classes is obvious.

In the rest of the paper, we essentially concern ourselves with the latter category.

2.2 *What are real-time systems*

There have been several dichotomies of systems such as deterministic/nondeterministic, synchronous/asynchronous, off-line/on-line, virtual time/real-time, sequential/concurrent etc. However, from the point of view of the basic philosophy of design, we can conveniently distinguish three categories of systems depending on the way the systems interact with their environment.

(a) *Transformational*: get the input in the beginning and send the output at the end.
(b) *Interactive systems*: interact at their own speed with users or with other systems.

(c) *Reactive systems*: maintain a continuous interaction with their environment, but at a speed which is determined by the environment (and not by the program or the system). In other words, the output may affect future inputs due to feedback. In general, we can further categorize these systems depending on the need for absolute time. However, in the sequel we will make the distinction explicit when required.

From the design point of view (a) and (b) have almost identical characteristics in the sense, these systems can be characterized by functions. However, the same is not true of reactive systems. A reactive system, in general, does not compute or perform functions (Harel & Pnueli 1985) but is supposed to maintain a certain ongoing relationship with its environment.

The dichotomies mentioned earlier are equally applicable to these systems. However, what we are interested in mainly is to see how the methods of design of traditional systems are not amenable for the design of reactive systems. For this purpose, let us look at some of the issues one is faced with in the development of a complex system.

2.3 *How to design a reactive system?*

The main issues one addresses in the development of a complex system can be broadly categorized as follows.

(a) The need for separation of concerns: that is, the question is, how does one decompose the behaviour of the system? This is an important issue as it provides a basis for system design.
(b) Refinement of behavioural components of the systems.
(c) Interaction of the behavioural components.

In the case of reactive systems, most of the times it is even difficult to provide some behavioural decomposition even if one ignores the necessity of the decomposition forming the basis for system design. In other words, the separation of concerns turns out to be extremely difficult. For example, even small real-time systems such as tactical embedded system for an aircraft might be simultaneously maintaining a radar display, calculating weapon trajectories, performing navigation functions etc. In these kinds of systems, one sees that (1) the code implementing the various tasks is mixed together such that it is difficult to determine which task(s) a given part of the code performs, and (2) the timing dependencies between code sections are such that changing the timing characteristics of one section may affect whether or not many otherwise unrelated tasks meet their deadlines. Thus, one of the challenges is to provide a method for behavioural description such that:

(i) behavioural description leads to separation of concerns;
(ii) the behavioural description captures the *what* part effectively in a compositional (incremental) way.

In the case of transformational systems, it is possible to decompose the system in a way reflecting the natural structure of the problem. However, such a decomposition is almost impossible since the interface between the system and the environment is complex, asynchronous, nondeterministic, highly parallel and distributed. In other words, the behavioral description is the main issue that makes it quite difficult from the traditional systems. Thus, one of the immediate needs is to come up with a

compositional (or modular) behavioural description of the real-time systems. This would provide a basis on which a sound methodology for real-time programming can be built.

3. Issues of modelling real-time reactive systems

As discussed already, a reactive system maintains continuous interaction with the environment and maintains a certain ongoing relationship with the environment. The three parameters that play a vital role in the modelling of real-time reactive systems are:

- communication mechanism;
- environmental abstraction;
- real-time.

Modelling of environmental abstraction is dependent on the actions that can be observed. In other words, models of concurrency for distributed systems provide a nice basis for environmental abstraction for reactive systems without the explicit notion of real-time. Thus, the issues of modelling real-time reactive systems can be broadly categorized into:

- models of communication;
- models of reactive systems;
- real-time and concurrency.

3.1 *Models of communication*

There are various ways of transmission from one task to another. Normally, in any mode of communication, one can identify three entities: sender, receiver and medium. The first two are active processes whereas the third is passive and it denotes the form of information in transit. A spectrum of media can be obtained based on various parameters such as:

- whether the sender can always send a message or can be blocked;
- the receiver may receive a message provided the medium is not empty;
- whether there is any constraint on the number of messages;
- whether the transmission is one-to-one or one-to-many etc;
- whether the order of messages received is the same as the order of messages sent etc.

It must be quite clear that a careful treatment of the media is essential for realizing a general model. Based on the various hardware architectures, one can obtain a variety of models. Some of the prominent ones are discussed below.

(1) The shared memory discipline broadly follows the following discipline:

- the sender may always write an item to a register or a location (of course, one assumes that the access of a register is mutually exclusive);
- the receiver may read an item from a register (or a location);
- reading and writing may occur in any order.

(2) One can treat the act of sending and receiving as one single indivisible act of communication; in other words, it can be treated as a point-to-point action. This is often referred to as *synchronous* or *handshake*. Such a discipline unifies the three

entities in the sense that sender and receiver participate in indivisible acts of communication experienced simultaneously by the sender and receiver. A further refinement can be obtained by defining whether the two participating agents exchange information with each other or the information flows in an unidirectional way. The equations for *handshake* communication will be discussed in the coming sections where the net effect of communication is replaced by a single non-observable action, denoted by τ referred to as the silent action or the perfect action. In fact, different models can be obtained by considering details such as whether the channels can be shared or the complimentary actions can be mapped to some other alphabet. Most of the formalisms of CSP (Hoare 1988), and CCS (Milner 1980), are built on these models.
(3) Another discipline, referred to as the *asynchronous discipline*, has the following characteristics:

- the sender is not blocked;
- the receiver can receive a message provided the medium is nonempty.

The asynchronous discipline can be further refined by:

- allowing simultaneous message transmission to various agents rather than point-to-point; this is often referred to as *broadcast transmission.*
- many agents can combine and exchange information among themselves; this is referred to as *multicast.*

A formal analysis of the broadcast mechanisms has been detailed in Shyamasundar *et al* (1987); the multicast paradigms can be seen in the *exchange functions* treated in Fitzwater & Zave (1977) and also in Shyamasundar *et al* (1991) in the context of formalizing coordinating actions.

3.2 *Models of reactive systems*

To make the discussion concrete, we take a very simple scheme for specifying behaviour of reactive systems without an explicit clock. A reactive system has an alphabet of events, corresponding to the set of possible observable[1] events in which the processes may be involved. The events require the participation of both the system and its environment which can be taken to be a user/another sub-component of a larger system. Following the notation of Milner (1980) and Hoare (1988), any reactive system is a well-formed syntactic term involving the events and the two operators + and ||. More precisely, given an event alphabet E, reactive processes are defined as follows.

(1) *nil* is a process which performs no actions.
(2) If p is a process then, for each $e \in E$, $e.p$ is a process which performs first an e and then behaves like p.
(3) If p, q are processes then $p + q$ and $p \| q$ are processes.

The operators + and || denote respectively the nondeterministic choice and the parallel composition operator. The operational semantics of the above language is given in terms of a labelled transition system.

A labelled transition system is given by (*Proc*, *Act*, $\rightarrow$) where *Proc* is the set of

[1] The role of observation in the modelling of real-time reactive systems is discussed while relating the models of reactive systems with time.

process states, *Act* is the set of actions (or events) and $\rightarrow \subseteq Proc \times Act \times Proc.\ p \rightarrow^a q$ is interpreted as follows: P is observed to do a, and changes to become q.

In sequential programming languages, there is a clear-cut notion of what the observable behaviour of a program is: an input/output pair leading to the semantic description of a program as a function or in the nondeterministic case, a relation or multifunction. However, for concurrent/reactive languages, there is no single canonical notion of observable behaviour but rather a multiplicity of semantic models. In particular, each notion of observability yields an answer to the following basic question which any semantics should address.

When do two expressions have the same meaning? That is,

$$p \sim q \Leftrightarrow \forall \text{ observable property } A: p\,sat\,A \Leftrightarrow q\,sat\,A.$$

In other words, two processes are equivalent (hence, have equal meanings) when they admit the same observations. By varying the notion of *observable property of behaviour*, we also vary the associated equivalence; the more we can observe, the more chances we have of distinguishing processes and, hence, fewer processes are made equivalent (equivalence is finer). Now, the transitions for the language defined above is given below: Let $\rightarrow$ be defined as the least relation satisfying the following axioms:

$$ap \xrightarrow{a} p$$

$$\frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'}$$

$$\frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'}$$

$$p \xrightarrow{a} p' \Rightarrow p \| q \xrightarrow{a} p' \| q$$

$$q \xrightarrow{a} q' \Rightarrow p \| q \xrightarrow{a} p \| q'$$

With the above language and the transition system, let us consider the various classes of observable behaviours that lead to different equivalences. Some of the presentation is based on the unpublished lectures by Abramsky (1989) and by Groote (1988).

3.2a *Traces*: Here, only sequences of actions can be observed; this is the simplest notion leading to the coarsest (reasonable) equivalence. Define,

$$p \xrightarrow{a} q, s = a_1, \ldots, a_n (a_i \in Act^*) \text{ iff } \exists p_1, \ldots, p_{n-1} \wedge p \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q,$$

$Traces(p) = \{s \in Act^* \mid \exists q, p \xrightarrow{s} q\}$. Then, the equivalence is defined by

$$p \sim_T q \Leftrightarrow Traces(p) = Traces(q).$$

The above equivalence is too coarse as it ignores the differences in deadlock behaviour. This is illustrated in the classical example shown in figure 1.

In the above example, $Traces(p) = Traces(q) = \{\varepsilon, a, ab, ac\}$. It may be observed that after doing a, p can always do b, while q can sometimes refuse to do b.

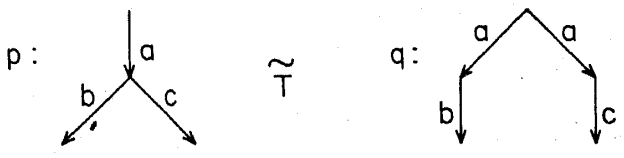

Figure 1. Trace equivalence.

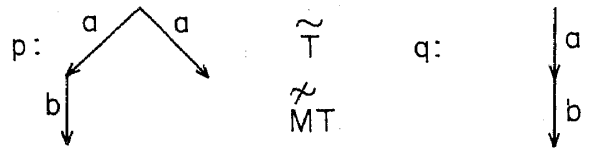

Figure 2. Complete trace equivalence.

3.2b *Maximal (complete) traces*: Consider the processes shown in figure 2.

It can be easily seen that $p \sim_T q$. However, it can be seen that the processes can in fact be distinguished by observing that process p has a trace a from which there is no further action and process q does not have such a trace. Refinements corresponding to such experiments can be obtained by adding the capability to observe *inactions*. Maximal (complete) traces of processes lead to such refinements.

Let $p \not\rightarrow$ denote the fact that there does not exist any a and p' such that $p \rightarrow^a p'$. Then, the maximal traces[2] (MT for short) are defined by,

$$\mathrm{MT}(p) = \{\sigma \mid \sigma \in A^*, \exists q, p \rightarrow^\sigma q \not\rightarrow\}.$$

Then the equivalence is defined by $p \sim_{\mathrm{MT}} q \Leftrightarrow \mathrm{MT}(p) = \mathrm{MT}(q)$.

3.2c *Failure sets*: If $s \in \mathrm{FT}(p)$ can be interpreted as: after process p does s, it refuses to do any action. A refinement of such a notion can be obtained by observing a process refusing a subset of actions offered by the environment. The equivalences that can be obtained with such an observable capacity are formally defined below.

Let $(s, \mathscr{X}) \in Act^* \times \mathscr{P}(Act)$. Define, $F(p) = \{(s, \mathscr{X}) \mid \exists q, p \rightarrow^s q \wedge \forall a \in \mathscr{X}, q \not\rightarrow^a\}$. Then the equivalence is defined by $p \sim_F q \Leftrightarrow F(p) = F(q)$

Consider processes p and q shown in figure 3.

It can be easily seen that $\mathrm{MT}(p) = \mathrm{MT}(q)$; however, $F(p) \neq F(q)$.

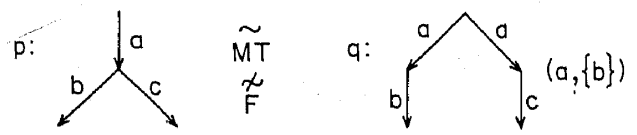

Figure 3. Failure set equivalence.

3.2d *Failure traces*: A refinement of the failure equivalence can be obtained by considering failure sets at all the intermediate points of the traces. Failure traces (denoted FT) is defined by,

$$\mathrm{FT}(P) = \{(a_0, \mathscr{X}_0)(a_1, \mathscr{X}_1) \cdots (a_n, \mathscr{X}_n) \mid \exists p = p_0, \cdots p_n, p_i \rightarrow^{a_i} p_{i+1}, \forall a \in \mathscr{X}_i, p_i \not\rightarrow^a, 0 \leqslant i \leqslant n\}.$$

Then the equivalence is defined by,

$$p \sim_{\mathrm{FT}} q \Leftrightarrow \mathrm{FT}(p) = \mathrm{FT}(q).$$

Consider the two processes shown in figure 4.

The two processes are not distinguishable under failure sets whereas they are distinguishable under failure traces. The latter follows from the fact that $(a, \{b\})ce \in \mathrm{FT}(p) - \mathrm{FT}(q)$.

---

[2]This is useful for terminating systems.

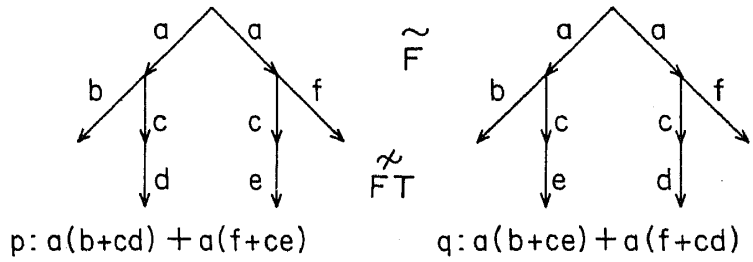

Figure 4. Failure trace equivalence.

3.2e *Ready sets*: If instead of observing what actions cannot be done, the enablement or the readiness of actions can be observed then one gets another notion of *ready equivalence*. Surprisingly, it turns out that this notion refines that of failure set equivalence. Define,

$$R(p) = \{(s, \mathscr{X}) | \exists q \cdot p \rightarrow^s q \wedge \forall a \in Act, q \rightarrow^a \Leftrightarrow a \in \mathscr{X}\}.$$

Then the equivalence is defined by,

$$p \sim_R q \Leftrightarrow R(p) = R(q).$$

Consider the processes shown in figure 5a.

It can be seen that the two processes shown in figure 5a are not distinguishable under failure sets whereas they are distinguishable under ready sets.

Note. It is of interest to note that from the set of ready pairs of a process graph[3], the set of failure pairs can be deduced but not the other way round.

Consider the processes shown in figure 5b. It can be seen that $p \sim^R q$; however, $p \not\sim^{FT} q$. Thus, from figures 5a and b, the incompatibility of failure traces and ready sets follows.

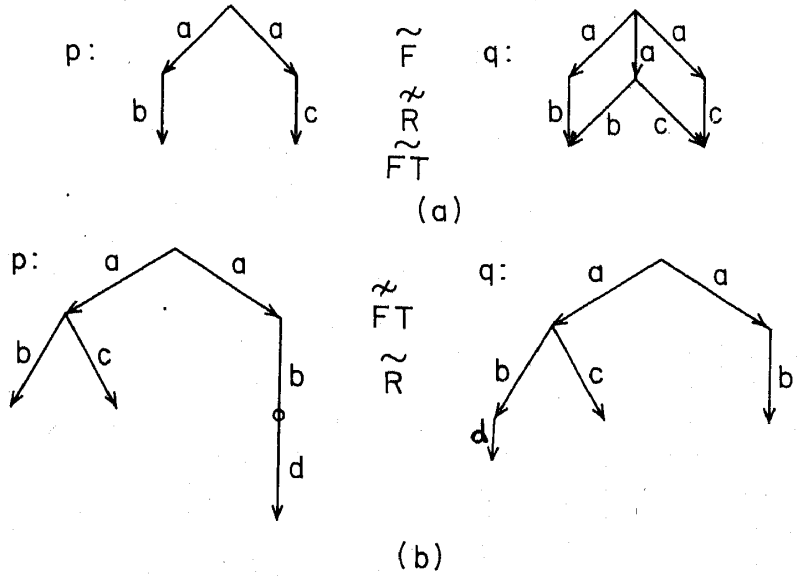

Figure 5. (**a**) Ready set equivalence. (**b**) Ready set and failure trace equivalence.

[3]In this paper, we have used these terminologies in an informal sense; for a formal definition, the reader is referred to Baeten & Weijland (1990).

3.2f *Ready traces*: The equivalence can be further refined by considering the ready sets at all the intermediate points. Ready traces (denoted RT) are defined by,

$$\mathrm{RT}(P) = \{(a_1, \mathscr{X}_1)(a_2, \mathscr{X}_2)\cdots(a_n, \mathscr{X}_n) | \exists p = p_0, \cdots p_n, p_i \rightarrow^{a_i} p_{i+1}, P_i \rightarrow^{a} \Leftrightarrow a \in \mathscr{X}_i\}.$$

Then the equivalence is defined by,

$$p \sim_{\mathrm{RT}} q \Leftrightarrow \mathrm{RT}(p) = \mathrm{RT}(q).$$

Consider the processes p and q as shown in figure 4. It can be easily seen that $p \sim^{R} q$ and $p \not\sim^{\mathrm{RT}} q$. The latter follows by looking at all the ready sets at the various points of the process graphs. Now, consider the processes shown in figure 5a from the point of view of ready and failure traces.

- $\mathrm{FT}(p) = clos\{(a, \{b, c\})(b, \{c\}), (a, \{b, c\})(c, \{b\})\}$;
  $\mathrm{FT}(q) = clos\{(a, \{b, c\})(b, \{c\}), (a, \{b, c\})(c, \{b\})\}$; hence, $\mathrm{FT}(p) = \mathrm{FT}(q)$, where *clos* gives the prefix-closure of the set with respect to the first element and any subset of the failure sets of the elements of the sequence.
- However, $\mathrm{RT}(p) \neq \mathrm{RT}(q)$, since
  $(a, \{a\})(b, \{b, c\}) \in \mathrm{RT}(q)$
  $\in \mathrm{RT}(p)$.

3.2g *Simulation and bisimulation*: In a sense, all the above equivalences are some refinements of traces or execution sequences. In the following, we define some notions that are based on the notions of the execution trees induced by the processes.

A natural notion of process equivalence can be obtained by formally interpreting a labelled directed graph as a process; we refer to such graphs as process graphs.

A *simulation* of process graph G_1 (say, $\langle V_1, E_1 \rangle$) by process graph[4] G_2 (say, $\langle \mathrm{V}_2, E_2 \rangle$) is a binary relation, $\mathscr{R}$ between their nodes satisfying the following condition ($\rightarrow^a$ can be treated as a reachability relation):

(i) $V_1 \subseteq Dom(\mathscr{R})$.
(ii) $(p, q) \in \mathscr{R} \supset \forall a, p \rightarrow^a p'$ *in* $G_1 \supset q \rightarrow^a q'$ *in* G_2, $\wedge (p', q') \in \mathscr{R}$.

Two graphs are *simulation equivalent (denoted* $\sim^S$) if there exists simulations in both directions.

It may be noted that R^{-1} need not necessarily be a simulation. If R^{-1} is also a simulation then one gets *bisimulation equivalence.* This is the finest reasonable equivalence which is not based on the traces or execution sequences. This is formalized below.

In a sense, bisimulation is the finest reasonable equivalence (could be considered as single step true concurrency) based on the notions due to D Park and R Milner. Intuitively, the equivalence corresponds to comparing states for equivalence recursively by the condition that every action of P has a matching action of q leading to an equivalence state and vice versa. Formally, it can be defined as follows,

$$p \sim^{B} q \Leftrightarrow \forall a[\forall p', p \rightarrow^a p' \Rightarrow \exists q', q \rightarrow^a q' \wedge p' \sim^{B} q' \wedge \forall q', q \rightarrow^a q' \Rightarrow \exists p', p \rightarrow^a p' \wedge p' \sim^{B} q'].$$

In other words, bisimulation identifies processes just when they unfold into the same (unordered) labelled trees.

---

[4]Here, V_1 and E_2 respectively denote the set of nodes and edges of G_1.

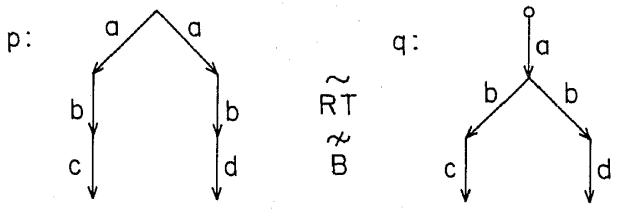

Figure 6. Bisimulation and ready trace equivalence.

Example. $a(b+c) \not\sim^B ab + ac$ since $b + c \not\sim^B b$ and $b + c \not\sim^B c$.

The concept of bisimulation can also be captured in terms of equations over sets as follows.

Define $Bis(p) = \{\langle a, Bis(q)\rangle | p \rightarrow^a q\}$,

Such a solution always exists if one assumes Aczel's anti-foundation axiom. That is, $Bis(p)$ may become a non well-founded set. The distinguishability of processes discussed above can be captured in terms of the following sets:

$$\langle a, \{\langle b, \{\langle c, \phi\rangle\}\rangle, \langle b, \{\langle c, \phi\rangle\}\rangle\}\rangle \in Bis(p)$$
$$\notin Bis(q).$$

The example shown in figure 6 illustrates that bisimulation refines ready traces.

A further refinement of equivalences lying between simulation and bisimulation can be obtained by refining the simulation equivalence. Two of such refinements are defined below.

A *2-nested simulation* is a simulation with the property that related nodes are simulation equivalent. In other words, there also exists a simulation in the reverse direction between the subgraphs that are rooted by related nodes.

Two graphs are 2-nested simulation (denoted $\sim^{2n-S}$) equivalent if there exists 2-nested simulation in both directions.

For the processes shown in figure 2, we have $p \sim^S q$ and $p \not\sim^{MT} q$. From this and the fact that trace equivalence is contained in simulation and maximal trace equivalences, we can conclude the incompatibility of the two. The incompatibility of the ready trace and simulation follows from the example shown in figure 7a.

A *ready simulation* is a simulation such that related nodes have the same set of initial actions. The underlying equivalence is referred to as ready equivalent and denoted by $\sim^{RS}$.

The examples shown in figures 7b and c differentiate the equivalences due to simulation, 2-nested simulation and bisimulation.

3.3 *Comparison of the various equivalences*

The following theorem (cf. Baeten & Weijland 1990 for proof) shows the implications of the various equivalences discussed above.

Theorem 1. *Let g and h be any two process expressions. Then,*

(1) *if* $g \sim^B h$ *then* $g \sim^R h$; (2) *if* $g \sim^R h$ *then* $g \sim^F h$; (3) *if* $g \sim^F h$ *then* $g \sim^T h$; (4) *if* $g \sim^B h$ *then* $g \sim^{RT} h$; (5) *if* $g \sim^{RT} h$ *then* $g \sim^{FT} h$; (6) *if* $g \sim^{RT} h$ *then* $g \sim^R h$; (7) *if* $g \sim^{FT} h$ *then* $g \sim^F h$.

Note. It may be noted that traces take account of the intermediate state in a very weak way whereas bisimulation does so in a very strong way.

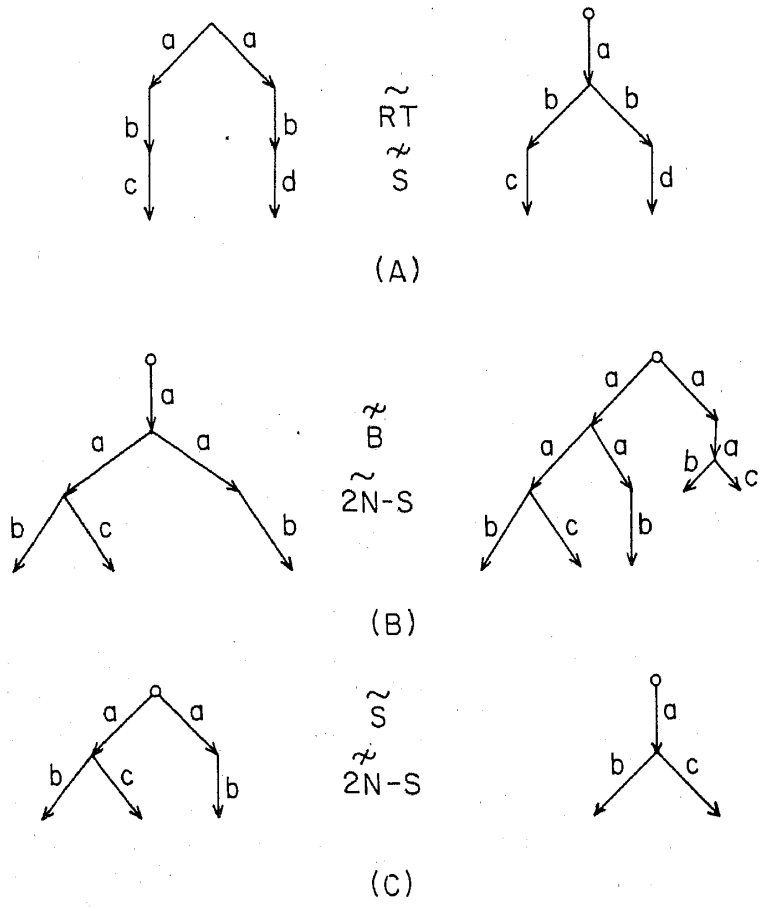

Figure 7. (**a**) Ready trace and simulation equivalence. (**b**) Bisimulation and 2-nested simulation. (**c**) Ready simulation and 2-nested simulation equivalence.

The results of the above theorem and the various incompatibility relations among the various equivalence notions illustrated earlier is nicely captured in the semantic lattice shown in figure 8.

We can summarize the various observational characteristics that make the various equivalences finer than the other equivalences as follows:

- observability of *inaction* refines maximal traces over that of traces;
- observability of *blocking* refines failure sets over that of maximal traces;
- if the observability of *blocking* is made dynamic then we get the failure traces;
- observability of the actions that a process can make gives the power to ready sets; the dynamisation of the ready actions leads to ready traces;
- on the other hand, giving the power of copying leads to simulation and the power of global testing leads to bisimulation equivalence.

3.4 *Observational and bisimulation equivalence*

Now that we know that bisimulation is strong and very nice from the point of view of equivalences, let us look at bisimulation from a computational point of view. An immediate question that arises is:

Is bisimulation based on a reasonable notion of observable behaviour? or is it too fine?

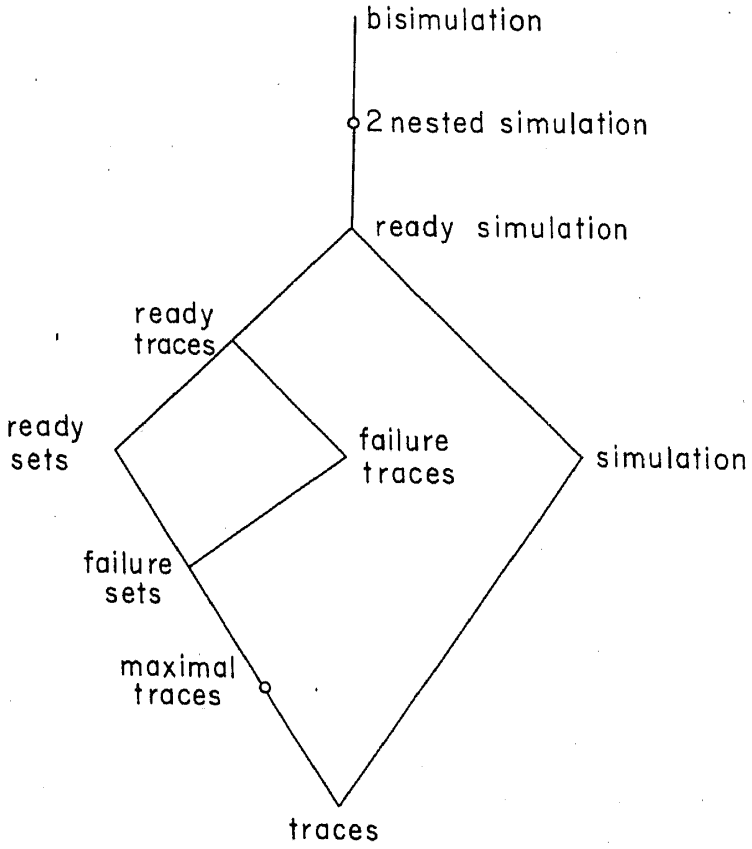

Figure 8. Relationship among various equivalences.

In other words, is there any way one could observe all the distinctions it makes by performing experiments? For example, consider the processes shown in figure 9.

Obviously, $p \not\sim^B q$. However, by performing experiments on the observable actions there is no way the two can be distinguished.

The question of looking at bisimulation from the point of view of the underlying traces has been addressed in Bloom *et al* (1988, pp. 229–239). They argue that the notion of trace congruence cannot be captured as a trace congruence of any "reasonable" process constructions. Larsen & Skou (1989) have defined the notion of probabilistic bisimulation to tackle the argument against bisimulation given in Bloom *et al* (1988, pp. 229–239). Groote & Vaandrager (1989, pp. 423–438) have studied the relation between bisimulation and structured operational semantics as well as the property of full abstractness. An attempt towards an unification of the frameworks has been envisaged in Abramsky & Vickers (1991).

3.5 *Other factors related to models of concurrency*

3.5a *Treatment of silent actions*: In the transition system given above, we have not said anything about the type of communication. A model of the simple *synchronous*

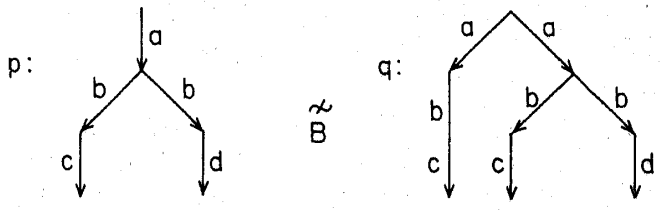

Figure 9. Bisimulation and observational equivalence.

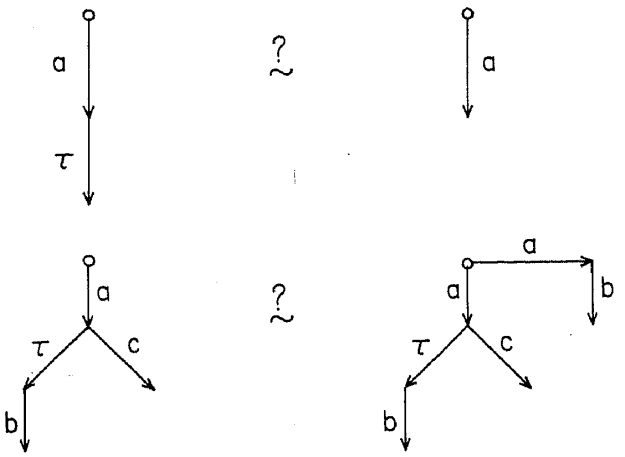

Figure 10. Effect of τ actions on equivalences.

communication can be obtained by adding the following axiom for our earlier transition system.

$$p \rightarrow^{a} p', q \rightarrow^{\bar{a}} q' \Rightarrow p \| q \rightarrow^{\tau} p' \| q'$$

Here, τ is referred to as the silent or the perfect action (Milner 1988). In a sense, we can now ask the question: *to what extent are the silent steps observable*? From the point of view of observation, one can ask questions like: can one observe silent actions *before* or *after* an observable event? For example, depending upon the choice of equivalence or inequivalence of the processes shown in figure 10, one gets different models.

3.5b *Linear time vs branching time*: In the previous sections, we have essentially considered two classes of equivalences:

- pure traces or refinements of traces;
- bisimulation.

The first class can be termed *linear time equivalences* in that a process is determined by its possible executions. The second class, that is bisimulation can be termed *branching time equivalence* which not only preserves the traces but also the branching structure of processes. One of the most popular arguments in favour of the branching time semantics was the fact that it allows a proper modelling of *deadlock behaviour* whereas the linear time does not. However, it can be seen from the various equivalences discussed that even though this is true for the case considering pure traces, the same comment does not hold in the context of ready or failure sets. In fact, an additional advantage of the linear time equivalences discussed above is that one also gets the notion of testing or observation for distinguishing processes. The main criticism of branching time structure is that distinction between processes are made that cannot be observed or tested, unless observers are equipped with extraordinary abilities like copying or global testing (cf. Abramsky 1987).

Even though bisimulation preserves the branching structure of the processes, an anomaly arises in the context of Milner's observational equivalence as illustrated (from Van Glabbeek & Weijland 1989) in figure 11.

It may be noted that in figure 11 (A), we have a path $a\tau b\tau c$ with outgoing edges $d_1, \ldots, d_4$, and it follows easily that all the three graphs are observation equivalent. It may be noted that b-edges (shown in broken lines) may be added without disturbing

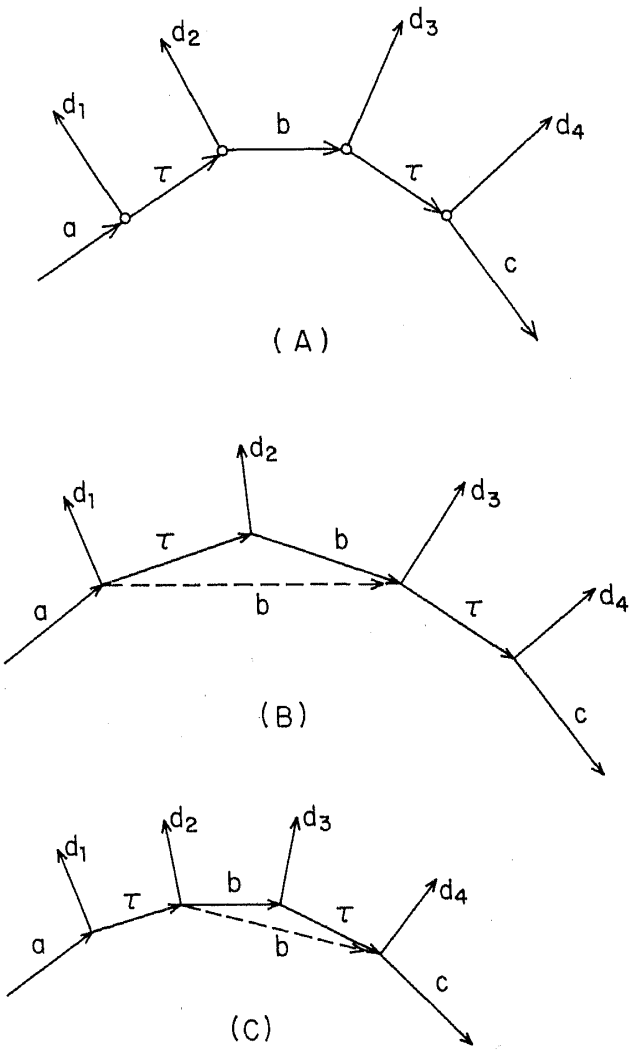

Figure 11. Observation equivalence in branching time.

the equivalence. However, in both (B) and (C), a new computation path is introduced in which an outgoing edge d_2 (or d_3 respectively) is missing; in fact such a path did not occur in (A). In other words, in the path introduced in (B) the options d_1 and d_2 are discarded simultaneously, whereas in (A) it corresponds to a path containing a state where the option d_1 is already discarded but d_2 is still possible. Further, in the path introduced in (C) the choice not to perform d_3 is already made with the execution of b-step, whereas in (A) it corresponds to a path in which this choice is made only after the b-step. From this it follows that observation equivalence does not preserve the branching structure of processes and hence lacks one of the main characteristics of bisimulation semantics.

3.5c *Interleaving vs true concurrency*: Two extreme ways of modelling concurrency are:

- concurrency is nothing but nondeterministic interleaving of concurrent events;
- concurrency is a phenomenon quite independent from nondeterminism.

Consider the process $a.nil \| b.nil$, which is specified to do the actions a and b concurrently. In the first view, the trace semantics of this process is given by

$$\{c, a.b, b.a\}.$$

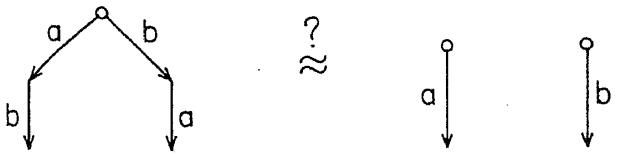

Figure 12. Interleaving vs. true concurrency.

Thus, in this model, this process is identified with another process $a.b.nil + b.a.nil$, which does actions a and b one after another but nondeterministically in either order. A typical model, that distinguishes concurrency from nondeterminism, is the partial order model. In this model, the above process is given by the poset $\langle \{a, b\}, \leqslant \rangle$, where the events a and b are unrelated by the relation $\leqslant$; in this model for actions $x, y, x \leqslant y$ means that x occurs before y and if two actions are unrelated, then it is not known in which order these actions take place.

The first model reduces concurrency to nondeterminism. As a consequence, any particular action in a process may be arbitrarily delayed. If other component process(es) involves an infinite number of actions then there is no upper bound on the time within which any action will be executed. In the worst case, an action may ever be delayed. In the second techniques, we have an extra "simultaneous" operator and two events will be related with each other if they are causal with reference to each other. The following trivial example shown in figure 12 illustrates the difference between the two informally.

3.5d *Treatment of divergence*: Another crucial point lies in the way infinite sequences of τ-steps in a process are treated. In the failure semantics proposed by Brookes *et al* (1984), all processes having an infinite τ-sequence from the root are set equal (to process CHAOS). For example, one can generalize such notions as by equating the following two processes as shown in figure 13.

The notion of bisimulation is more discriminating. The advantage is that process models obtained by bisimulation equivalence satisfy useful abstraction principles based on fairness. For example, Koomen's fair abstraction rule gives a way of simplifying processes by elimination of (some) infinite τ-sequences. This elimination can be understood as fairness of (visible) actions over silent τ-steps.

3.6 *Concurrency and real-time*

From the survey on concurrency, it must be apparent that the concurrency theory can be seen as an abstraction of observation. In a sense, for a natural and a formal abstraction of distributed systems it is necessary that theories must take into account the physical laws that distributed systems must obey. One of the prime factors that must necessarily be tackled is the relation between *logical time* and *physical time* for

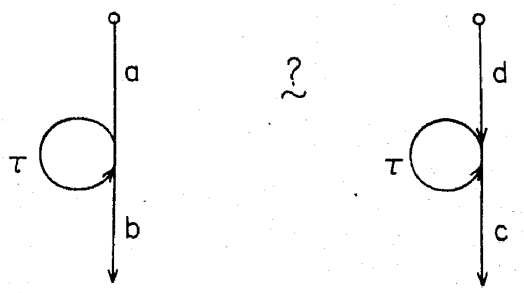

Figure 13. Effect of divergence on equivalence.

the understanding of real-time distributed systems. It is a standard paradigm of physics to understand the notion of atomicity of a thing or explore the internal structure of aspects previously considered as atomic. Thus, an observation in the context of concurrency in the presence of *real-time* necessitates the understanding of an event and atomicity. In the following sections, we provide a background on the notion of an event and the notion of atomicity and discuss the various time domains that have been used in the specification of real-time distributed systems. With such a background, we discuss the possible choice of the various concurrency models in the context of time as an observable entity.

3.6a *Events and time domains*: One of the fundamental notions that needs a careful examination is the notion of an event. In fact, it is from such an analysis one can capture the notion of *observational behaviour* for real-time reactive systems. This problem has been nicely dealt with in Lelann (1983) with respect to the Newtonian and relativistic notions of observation.

Based on the notions of observability one of the immediate questions that arises is: *Is an event atomic*? If we consider an event as being something instantaneous that exists or not, then the question does not arise. Obviously, an invariant universe would not define time, and would not need it. It is only because states change that time acquires meaning. Thus, there is a need to consider some physical universe, $\mathcal{U}$, which includes elementary entities. Every entity will be associated with a set of states. An entity can only be in one state at a time.

Without loss of generality, we need only two states, say *true* and *false*. We will be interested only in the changes that bring an entity from state *false* to state *true*. Reaching state *true* is what constitutes an instantiated event in universe $\mathcal{U}$. By definition of an event, entities cannot be observed while states are being changed. Consequently, an elementary entity state change $[false \rightarrow true]$ is the smallest atomic operation one can conceive in $\mathcal{U}$ (symbol $\rightarrow$ reads *precedes*). Two successive state changes $[false \rightarrow true]$ for a given entity in $\mathcal{U}$ correspond to two infinitesimally close points in $\mathcal{U}$'s spacetime. Now, time can only be defined and instantiated as a change of state occurring at some location, e.g., raise of a pulse on a wire, or a division on a clock face etc.

In trying to define time for an instantiated event, we find ourselves back in considering timeless events. We are then forced to admit that definition of an event in some real universe is meaningful only if we assume that it is possible to observe concurrent phenomena unambiguously in this universe, or, in other words, that the ordering of the termination operations is an invariant in this universe. One of these two operations is instantiated as a *physical clock* state change denoted $t \rightarrow t\ next$. Assume that the physical clock is in location k. Assume the other operation takes place in some other location ℓ.

A change $[false(\ell) \rightarrow true(\ell)]$ is said to be an event (ℓ, t) if

$$\{false(\ell); t(k)\} \rightarrow true(\ell) \rightarrow t\ next(k).$$

The relative ordering of $false(\ell)$ and $t(k)$ does not matter. For such an event to be observed unambiguously in universe, $\mathcal{U}$, it is necessary to assume that every change of $t(k)$ is communicated instantaneously to all entities in $\mathcal{U}$. In practice, this entails the following two requirements:

- $t(k)$ is communicated with a delay that is negligible compared with the interval separating two consecutive state changes;

- $t(k)$ is communicated almost simultaneously to all entities, the time dispersion for any two entities being negligible compared with the interval separating two consecutive state changes.

Whether such requirements can be satisfied depends entirely on $\mathcal{U}$'s spacetime topological properties. When we do not know how to achieve appropriately timed broadcasting of a unique signal on different physical paths, we are left with the problem of dealing with propagation delays that are not negligible compared with clock periods and that may vary at different instants over some given physical path. These are the conditions for adopting a relativistic view of time.

Two approaches are possible, depending on $\mathcal{U}$'s spacetime topological properties:

- Approximate some unique time dimension throughout $\mathcal{U}$ via the definition of a spacetime-independent transformation F; under specific timeless assumptions (e.g., existence of finite lower and upper bounds for propagation delays), one can devise algorithms for which proofs establish that the relative drift[5] of any two clocks will never exceed some "acceptable" value. We are then dealing with Newtonian Physics. Most real time distributed computing systems of limited scale fall into this category.
- Correlate different time dimensions with each other throughout $\mathcal{U}$ via the definition of a spacetime-dependent transformation F (e.g. Lorentzian transformation), when specific timeless assumptions cannot be made. For example, the situation created by a clock signal that travels faster (respectively slower) than expected can be equated with a situation where the receiver of the signal (respectively the sender) is being accelerated relative to the sender (respectively the receiver). The same situation arises when the relative motion of the clocks or gravitational effects are not negligible, as exemplified by the Global Positioning System (Navstar). We are then dealing with Relativistic Physics.

Clearly, the problems that are derived from the presentation above cannot be avoided by utilizing extremely accurate clocks, as is sometimes believed. Caesium clocks which have a timekeeping capability of 0·1 μs/day would be useful for very closely approximating the implicit statement that all clocks behave identically in identical circumstances. But even such good clocks cannot influence properties of signal propagation delays.

We have assumed so far that a state change [*false* → *true*] is the smallest atomic operation one can think of. But how is atomicity effectively obtained? One could imagine that specific elementary physical devices could be built that would implement, say at the bit level, these atomic state changes. Unfortunately, there are a few problems which prohibit us from assuming that such is the case. For example, as the levels of energy and speed of signals are never infinite in computing systems, state changes are not instantaneous. While a state is being changed (Write) many observations (Reads) can take place. These operations are not mutually exclusive. Reads which observe internal states violate atomicity requirement that states internal to an operation must be kept visible to other concurrently executing operations. One could let each Read choose more or less randomly which final good state has been supposedly observed ("0" or "1" for example). It seems that we have a solution. However, we see that if we want to state properties about schedules or Reads which are concurrent

[5] Some of these aspects have been discussed in Koymans *et al* (1988).

with one single Write, we must assume that some final good state is eventually reached (that is atomicity requirement A1) and that an algorithm exists whereby Reads which have been truly concurrent with the Write are viewed as being uniquely ordered. Such algorithms are available in the literature. We are left with the problem of deciding how to guarantee that a final good state is eventually reached. Hardware designers of conventional circuits have faced this problem for many years. Oscillatory and metastable states can be entered for undetermined times by such simple devices as flip-flops. Identical problems are encountered by VLSI circuit designers.

All proposals which have been made to circumvent the problem consist of enforcing the existence of an upper bound for state change durations or observations. A similar requirement is necessary for the algorithms given towards synchronization of clocks. Can we say that such proposals do not assume a lower-level physical solution to the initial problem? No, in the sense that specific timing properties must be assumed for the basic operations or state changes.

Again, time underlies the concept of atomicity. Problems of time are dealt with explicitly by hardware designers and designers of real-time systems. These problems are in fact very general and should be carefully addressed in every system design. Many of such assumptions can be seen in the spectrum of real-time languages surveyed in Shyamasundar (1991a, b).

In the following section, we consider time domains used in the modelling of real-time systems.

Time domains – In the linear time and branching time[6] models (and other models) even though the term *time* is used, a very restricted notion of time is used, which is not satisfactory for real-time systems. In these models one can only say that whether two events took place at different points of time or not. But for modelling real-time systems, we need to know the exact times at which various events take place. A straightforward way of incorporating a notion of time is to associate with each event, the time at which that event takes place. An immediate question that arises is: what are the values time should take? There are a number of proposals:

- the time values are integers;
- time ranges over *real* values;
- time ranges over a total order in which a distance metric can be defined.

Proponents of integer time argue that the systems being modelled are discrete systems and hence we need to consider only discrete integer values. Though this is a good assumption for synchronous models, the claim does not hold in asynchronous systems in which different events can take place at points that are arbitrarily close to each other. In such systems, the right time values one should use are real values or at least values from an Archemedian field.

Concurrency complicates modelling real-time systems: should one use same or different clocks for events happening in distinct sub-components? Most of the models make the simplifying assumption that there is a global clock according to which different events happen. This assumption is not very different from the one in which each concurrent subsystem has its own clock but with a definite relationship between

[6] In fact, if we consider real-time events the same actions on different branches may not be the same; this would have to be integrated with real-time aspects of the models.

the time shown by different systems. In some highly distributed systems involving autonomous components, the latter assumption is not valid. But one can not do reasonable real-time computing without assuming any relationship between the clocks of different subsystems. A logic of concrete time intervals has been developed in Lewis (1990, pp. 380–389) wherein time delays between the scheduling and occurrence of the events that cause state changes are constrained to fall between fixed numerical upper and lower time bounds. Such an abstraction is shown to be useful in the modelling of asynchronous systems.

3.6b *Choice of concurrency model*: Based upon the various parameters of concurrency discussed above, one can get a spectrum of semantic models. Broadly, the various models can be categorized into two distinct classes:

(1) interleaving;
(2) true concurrency.

The interleaving model is simpler as it reduces concurrency to nondeterminism. Also this allows uniprocessor implementation of concurrency. In contrast, the second view adds an extra parameter to modelling reactive systems and hence is less simple. But it is claimed that it is the right view in decentralized systems involving autonomous components as there is no notion of a global ordering of events.

Both these views are unsatisfactory for real-time reactive systems: in the first view, an event in a process may be indefinitely delayed while in the second view it is not known whether two concurrent events are executed simultaneously or at different times; it is not even clear whether one can relate the times of occurrences of two concurrent events. In fact, many of the models based on true concurrency suffer from the drawback that it either enforces complete synchronicity in executions or does not exclude interleaving. For real-time systems, we need a model that does not allow arbitrary delaying of event occurrences and that describes whether two events are executed simultaneously or not. One such notion is the *maximal parallelism* (Salwicki & Muldner 1981). Based on such a notion, a compositional model for real-time has been advocated in Koymans *et al* (1988). Such a model is *realistic* in the sense that concurrent actions can and will overlap in time unless prohibited by synchronization constraints; in other words, no unrealistic waiting of processors is modelled. The following examples illustrate the intuitive ideas behind this model.

Simple shared variable model – Consider the following program:

$$[P_1::x:=1 \,\|\, P_2::x:=3 \,\|\, P_3::y:=2].$$

Let us assume that multiple accesses to a single (*shared*) variable are mutually exclusive. Then in the above program, either P_1 and P_3 or P_2 and P_3 will execute their first move *simultaneously*, but not P_1 and P_2.

Distributed program(CSP-R): Consider the following program[7]:

$$(P_1::P_{11}::P_2!0 \,\|\, (P_{12}::P_{13}!1 \,\|\, P_{13}::P_{12}?x; P_2!x).$$

[7] Here, $P_1?x$ in P_2 denotes the waiting of P_2 for receiving a communication from P_1; similarly, $P_1!e$ in P_2 denotes that the process is waiting for sending a value e to process P_1; on handshaking, $P_1!10 \| P_2?x$ results in x being assigned e.

According to the *interleaving* semantics the following two scenarios are possible:

(1) P_{11} communicates with P_2 while P_{12} communicates with P_{13}; after that P_{13} communicates with P_2.
(2) P_{12} first communicates with P_{13} followed by P_{13} with P_2; finally P_{11} communicates with P_2.

According to the *maximal parallelism* semantics, only (1) is possible since P_{11} and P_2 can immediately become involved in a *handshake* and hence do not wait for P_{12} and P_{13}. In other words, in the distributed computing the maximal parallelism can be interpreted to mean *first-come-first-served.*

Now, let us see how we can describe the maximal parallelism semantics for our simple language described earlier. Let us assume that the execution of all basic actions takes the same amount of time.

$$\frac{p \xrightarrow{a} p', q \xrightarrow{b} q'}{p \| q \rightarrow^{\{a,b\}} p' \| q'}$$

$$\frac{p \xrightarrow{a} p', q \not\rightarrow}{p \| q \rightarrow^{\{a\}} p' \| q}$$

$$\frac{p \not\rightarrow, q \xrightarrow{b} q'}{p \| q \rightarrow^{\{b\}} p \| q'}$$

In other words, a process cannot wait unnecessarily. Since, enablement implies that there must be a processor for every process, one can see that the basic maximal parallelism model assumes that there are as many processors (machines) as there are parallel components in the system. This assumption can however be relaxed by relaxing the requirement that event should occur not immediately but with bounded delay. These aspects have been addressed in Koymans *et al* (1988). In fact, it is also possible to relax the requirement of one processor for every process by modelling scheduling (in a restricted manner these have been addressed in Liu (1989) and Liu & Shyamasundar (1990, pp. 21–26)).

Now, let us analyse the question: *Does maximal parallelism provide a good model for real-time systems*? Though the model is realistic in a sense it suffers from some conceptual problems. This is illustrated by the following example illustrated in figure 14(A).

Consider a network with distributed control, and two processes A and B in different nodes that want to communicate with a process C in a third node. If A wants to

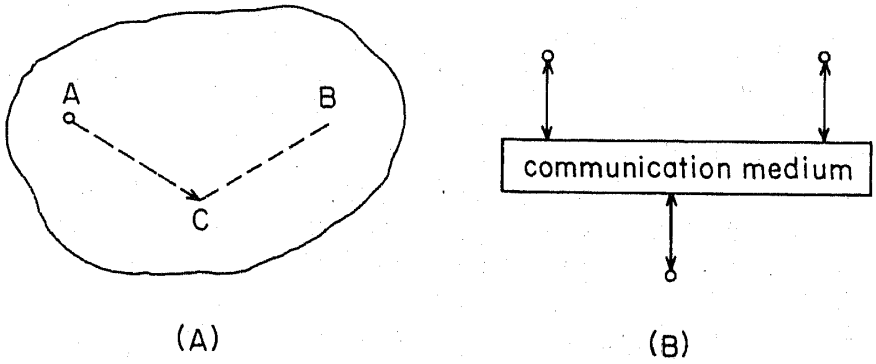

Figure 14. Effect of topology and communication medium.

communicate at an earlier time than B, relative to some global time scale, then according to the *first-come-first-served* (*fcfs*) principle, indeed, A should communicate first. Whether A's message *arrives* in C before B's message or not, depends on the topology of the network. So, imposing an *fcfs* principle upon the order of communications induces non-trivial requirements upon an underlying communication layer requirement that we would like not to make. Similar problems occur if processors communicate, for example, via a common bus where assumptions about bus-arbitration have to be taken into account. The lesson that should be drawn from this example is that, whereas the maximal parallelism model applies the *fcfs*-principle to the order of initiation of requests, the principle should rather be applied to the order in which a process becomes aware of requests. In doing so, we create the freedom to relax the stringent impositions of the original model on the behaviour of a communication layer. Specifically, in this way it becomes possible to vary the time-gap (0 in the original model) between the initiation and receipt of a communication request, which reflects the uncertainties about the communication layer. This variation of the time gap is the essential feature of the $MAX_\gamma(\delta, \varepsilon)$ model of distributed concurrency. The parameters δ and ε function as lower and upper bounds on the above time gaps which are allowed to take on any value in between these bounds. As a consequence, communications that are initiated too close in time (relative to the global clock) cannot be temporally ordered anymore. These time bounds may be interpreted as an abstraction of the propagation delays within some communication layer. The third parameter, γ, of the model is used to extend communications in time and denotes the number of time units it takes.

In the $MAX_\gamma(\delta, \varepsilon)$ (see figure 14B), it is assumed that there is no unnecessary waiting between the execution of actions. Communication between processes is served on a first-come-first served basis. Additionally, the following model pertains to process-communication:

- processes communicate via a medium.
- it takes between δ and ε time units (ε not included) for the medium to become aware of a process expressing its willingness to communicate or withdrawing its willingness (time-out).
- communication between two processes only occurs after the medium has become aware of both processes' willingness.
- a communication takes an additional γ time units during which period the processes remain synchronized.
- a communication that is in progress at a time when the medium receives a time-out from one of the participating processes, will be completed; a communication that might be started at such a time, will not be executed.

The formal details of these models are discussed in Koymans *et al* (1988).

There is another model of reactive systems referred to as the *strong synchronous model* that is useful when there is no need of explicit clock. According to strong synchrony hypothesis, any event, be it a communication event between two distant machines or a local event occurring within a single machine, takes place instantaneously. Obviously, this hypothesis is not valid for large systems extended in space. However, this is a very useful simplifying assumption for embedded systems occupying small space.

Having chosen a concurrency model, the other important aspect that needs to be looked into is: To what extent should the real-time aspects be incorporated in the given concurrency model. An important question that arises is:

Is there need for a special status of time or is it another parameter of the state? In fact, such a question has already been studied in the modelling of dynamic systems where a special status is accorded to time (based on which, one gets various classes of equations). In the same way, even in the context of programming, it appears necessary to accord a special status to the "time" parameter in order to specify various real-time properties. It may be noted that the parameter "time" is already distinguished by the fact that it is continuous, monotonic and divergent.

On the whole, a real-time model for concurrency depends upon:

(1) what can be specified/proved in the given model of time?
(2) what is the complexity of the decision procedures?
(3) what is the relation of the established property to the physical system property?

4. Challenges in the design of real-time systems

The challenges that underlie the design of real-time systems can be broadly categorized into:

- specification and verification of real-time programs;
- real-time programming languages;
- systematic development of real-time programs;
- real-time scheduling;
- tools for the design of communication protocols.

In the following, we discuss these aspects in detail.

4.1 *Specification and verification of real-time programs*

Specification formalisms are central to the problem of developing safe reliable real-time distributed systems. Handling real-time will not only require the development of specification and development frameworks but might also require a revision of the basic models that have been used so far in dealing with concurrency. One of the main goals of any specification formalism would be to bridge (or narrow) the gap between specification and implementation. The next question is: What are the general properties for any candidate formalism? Obviously, the formalism should support *compositional verification.* That is, it should be possible to verify the specification of a program based entirely on the specification of its constituent components without looking into the interior structure of the components. In fact, it is preferable to support the stronger notion of *modularity* (cf. Zwiers 1988)[8]. Of course, any automated (even partial) support environment would be a welcome feature of any method for the design of a complex system. Generally speaking one should address the following questions:

- Given a model, find the most suitable formalism in which to express a given property.

[8] For compositionality, one requires that from a given complete program specification it is necessary to establish the existence of specifications of the components from which the complete specification can be deduced. However, for modularity, one has to establish that a deduction of the complete program (or specification) is possible from a given a priori specification of the components. Needless to say, the latter goes naturally with the philosophy for the design of large programs.

- For deriving manageable verification techniques, it is necessary to build a tradeoff among ease of expression, generality and amenability. It may be observed that the more general the specification the easier it is to specify; however, the associated verification method will become harder. As in any area, researchers have considered subsets of the general problem and devised nice techniques (for a survey, see Shyamasundar 1991a, b). We have already seen the various issues of modelling real-time systems in the earlier sections.

Most of the existing works make the simplifying assumption that there is a global clock according to which different events happen. This assumption is not very different from the one in which each concurrent subsystem has its own clock but with a definite relationship between the time shown by different systems. In some highly distributed systems involving autonomous components, the latter assumption is not valid. But one can not do reasonable real-time computing without assuming any relationship between the clocks of different subsystems. Coming to modelling, the approach of modelling concurrency via nondeterminism can be immediately ruled out from considerations of predictability. However, it is necessary to model the nondeterministic environment. The work on real-time systems can be broadly divided into the following two streams:

(i) *Strongly synchronous systems* – Here, interaction between the components of the systems as well as the environment is synchronous and instantaneous, control or communication does not take any time, and further, there is explicit notion of *clock*. Further, nondeterminism is completely ruled out. In a sense, the focus here is on ideal system behaviour as in some parts of engineering and mathematics.
(ii) *Asynchronous distributed systems* – Here, the interactions are asynchronous and take arbitrary but bounded (it can vary between some limits) time.

But in practice, systems are neither purely synchronous nor purely distributed. Some layers (parts) of the systems will be synchronous while certain other layers (parts) will be asynchronous. A robot is a typical abstraction of such a system. A robot consists of a number of sensor/actuator components – one for each of its hands and legs, a sensor to see, a sensor to hear and so on. Each of these sensors is localized and, hence, a strong assumption about synchronicity is viable. In order for the robot to do globally meaningful tasks (like moving around space avoiding obstacles, moving objects from one place to another) all the sensors in its body will have to interact with each other. Since these sensors are distributed over the entire body of the robot the communication delay between them will be appreciable and cannot be ignored. Hence the interaction between the sensors will have to be modelled by asynchronous communication. Further, in the modelling of real-time systems it becomes necessary to model nondeterminism due to the environment; note that it should also be possible to capture the predictable (does not necessarily mean deterministic) requirements of the real-time systems. In short, a unified integrated approach of strongly synchronous and asynchronous/synchronous will provide a nice formalism for the specification of real-time distributed systems. Such an approach will also throw light on unification of the various theories of concurrency. It may be noted that the unification also requires refinements of the semantics/the proof theory of the strongly synchronous and asynchronous distributed systems.

4.2 *Real-time programming languages*

One of the main goals of a programming language is to provide a natural vehicle for expressing good ideas elegantly. However, if we look at a large spectrum of real-time languages, the languages do not reflect any evolution with respect to assembly languages. However, the scene is changing rapidly and low-level programming techniques will not remain acceptable for large safety-critical systems (cf. Berry 1989). Real-time programming will follow the modern tendency to make systems hardware independent: *software has a longer lifetime than hardware*. Some of the main issues of research are:

(1) expressibility of timing requirements;
(2) exception handling mechanisms;
(3) efficiency;
(4) formal semantics and verifiability – it is important to consider realistic models of communication, concurrency and time. Further, the semantics must account for resource limitations in a natural way;
(5) an integrated environment for the development of real-time programs – from the point of view of reliability and robustness, it is very essential to provide analysis tools for timing and functional analysis of the components. In fact, mechanical support (with possibly graphical support) is very necessary for the wide acceptance of any language for programming large systems;
(6) reliability and fault tolerance of programs – it is important to obtain a proper tradeoff between hardware and software to cater to a variety of applications;
(7) object-oriented paradigm – as discussed already, flexibility is one of the most important factors in the design of real-time systems. Object-oriented programming perhaps would provide a good insight into these aspects. Broadly, an object-oriented program consists of objects and methods. An object may ask for methods defined in it or in other objects. In other words, one can define methods based on various criteria (perhaps including performance criteria) and the system can call the appropriate methods based on the need. Such a design will go a long way in providing a basis for portability satisfying timing constraints and would support even bottom-up techniques of building systems. Shyamasundar *et al* (1991) have shown formally that object-oriented programming is viable for real-time.

4.3 *Systematic development of real-time programs*

A sound methodology should enable one to arrive at a correct real-time program from their high level specifications. It must however be noted that from the point of view of deriving correct implementations from a specification it is just not sufficient to concentrate solely on functional or the temporal requirements. The possible implementations of a real-time system are quite often restricted by the configuration and resources of the execution mechanism that will be used to run the system. Thus, in order to judge the feasibility of the implementation derived from the specification it is necessary to formalize the properties of the execution mechanisms that will be used to run the system. Hence, apart from temporal requirements, paradigms of real-time systems also have to express implementation specific characteristics such as:

(i) multiprocessor/microprocessor/sequential, (ii) scheduling policies such as fixed priority, dynamic priority, round robin, time slicing etc. and (iii) the mechanism for the interaction with the environment such as interrupts or polling. That is, the high level specifications will state the timing properties and other implementation characteristics or properties of the programs being developed, while the final programs derived using the methodology will be the ones satisfying these constraints. In fact, transformational methodology would have all the advantages of the traditional stepwise refinement methodology. To find the right level of abstraction for describing the implementation-specific characteristics, it is essential for deriving implementations from specifications. This is a major research problem.

4.4 *Real-time scheduling*

One can view a real-time system as a set of tasks or as a set of periodic and sporadic processes. Thus, it is very essential to use efficient scheduling strategies for meeting the resource and timing constraints. Most of the scheduling algorithms have the following drawbacks:

(1) most of them are intractable, or
(2) most of the algorithms require that the component characteristics be known a priori and limit themselves to uniprocessor/multiprocessor configurations.

However, a large spectrum of process control tasks are inherently distributed with several hard real-time constraints. Thus, it looks imperative to look for scheduling algorithms (cf. Stankovic 1988) with good heuristics to derive efficient scheduling algorithms in the context of parameters such as (i) static vs dynamic scheduling, (ii) centralized vs distributed systems, (iii) hard vs soft deadlines, (iv) preemptable vs non-preemptable tasks, (v) fault tolerance etc.

4.5 *Tools for communication protocols*

Typically, real-life protocols can be considered to be a coordinated set of simple programs that are often time-dependent. There have been nice formalisms such as LOTOS for specifying protocols and workbenches for verifying them. However, the formalisms lack the power to

(1) express timing constraints such as minimal, maximal, durational etc;
(2) specify interrupts and priorities.

These features are very essential since *predictability* is an important aspect of protocols. These features can be seen in the language RT-CDL (Liu & Shyamasundar 1989, pp. 21–26) designed from the point of view of modelling general real-time reactive systems. With the ever-increasing use of protocols in various walks of life, it is important to arrive at formalisms that enable the overcoming of the above drawbacks. In fact, any formalism for protocols should be supported by a nice set of tools that enable the users to formally derive and verify them. It may be noted that the protocols are not necessarily finite state. However, a large class of them are finite state. Thus, in developing automated tools, it is necessary to look into aspects of how much of the non-finite state systems can also be handled.

5. Conclusions

In the previous sections, we have articulated real-time systems as *systems that maintain a temporal relationship with an uncooperative environment.* We have discussed the various issues of modelling concurrency, time and communication together and shown the various possible process equivalences. The choice depends on the observable entities and also on the application.

Further, we have argued that real-time systems have posed a wide spectrum of challenges to the computing community and highlighted the challenges in building real-time systems. To meet the challenge it is very essential to crystallize the behavioural model of real-time systems using realistic models. To sum up, one of the most immediate needs is the discovery of specification formalisms that can be embedded in an hierarchical method of refinement. Of course, for the success of a sound methodology it is very essential to arrive at a proper tradeoff among the notions of time, engineering limitations and physical abstractions. From an engineering point of view, there is a need to strike a nice balance between an ideal system and an actual system to derive a nice methodology for designing real-time systems.

Many ideas presented owe their origin to a large number of people. Many ideas have become clear during discussions by the authors with various people and are too numerous to mention. The authors thank all of them. The authors would also like to thank K Narayana Kumar for a careful reading of the manuscript.

Partial support by the Indo-French Centre for the Promotion of Advanced Research/ Centre Franco-Indien Pour la Promotion de la Recherche Avancee as part of the project "Formal Specification and Development of Real-Time Reactive Programs" is gratefully acknowledged.

References

Abramsky S 1987 Observation equivalence as a testing equivalence. *Theor. Comput. Sci.* 53: 225–241

Abramsky S 1989 Tutorial on concurrency, Unpublished lectures at the Principles of Programming Languages

Abramsky S, Vickers S 1991 *Observational logics and process semantics* (forthcoming)

Baeten J C M, Weijland W P 1990 *Process algebra* (Cambridge: University Press)

Bloom S, Istrail S, Meyer A R 1988 Bisimulation can't be traced: Preliminary Report, 15th ACM Annual Symposium on Principles of Programming Languages, San Diego, pp. 229–239

Berry G 1989 Real-time programming: Special purpose or general purpose languages. *Information Processing'89* (ed.) G X Ritter (IFIP, North-Holland Publishing Co.) pp. 11–17

Brookes S D, Hoare C A R, Roscoe A W 1984 A theory of communicating processes. *J. Assoc. Comput. Mach.* 31: 560–599

Fitzwater D R, Zave P 1977 The use of formal asynchronous process specifications in a system development process. *Proc. 6th Texas Conf. Computer Systems*, University of Texas at Austin, 2B-21:2B-30

Van Glabbeek R J, Weijland W P 1989 Branching time and abstraction in bisimulation semantics. *Int. Fed. Inf. Process. Congress' 89* pp. 613–618

Groote J F 1988 Tutorial on ACP, Unpublished lectures at the Workshop on Logic and Models of Concurrency, Bangalore

Groote J F, Vaandrager F 1989 Structured operational semantics and bisimulation as congruence. *ICALP'89 Lecture Notes in Computer Science. Vol. 352* (Berlin: Springer Verlag) pp. 423–438

Harel D, Pnueli A 1985 On the development of reactive systems. In *Logic and models of concurrent systems* (ed.) K R Apt, Nato ASI Series (Berlin: Springer-Verlag)

Hoare C A R 1988 Communicating sequential processes. *Commun. ACM* 21: 666–677

Koymans R, Shyamasundar R K, de Roever W P, Gerth R, Arun-Kumar S 1988 Compositional semantics for real-time distributed computing. *Inf. Comput.* 79: 210–256

Larsen K G, Skou A 1989 Bisimulation through probabilistic testing. *ACM Symposium on Principles of Programming Languages, Austin* (New York: ACM Press) pp. 344–353

Lelann G 1983 On real-time distributed computing. *Int. Fed. Inf. Process '83* pp. 741–753

Lewis H R 1990 *A logic of concrete time intervals. LICS Vol. 90* pp. 380–389

Liu L Y 1989 *Paradigms for the specification, design and verification of real-time distributed systems*, Ph D thesis, Pennsylvania State University

Liu L Y, Shyamasundar R K 1989 RT-CDL: A real-time design language and its semantics. *Int. Fed. Inf. Process '89* pp. 21–26

Liu L Y, Shyamasundar R K 1990 Static analysis of real-time distributed systems. *IEEE Trans. Software Eng.* SE-16: 373–388

Milner R 1980 *A calculus of communicating systems. Lecture Notes in Computer Science. Vol. 92* (Berlin: Springer Verlag)

Milner R 1988 *Communication and concurrency* (New York: Prentice Hall International)

Salwicki A, Muldner T 1981 *On the algorithmic properties of concurrent programs. Lecture Notes in Computer Science. Vol. 125* (Berlin: Springer Verlag)

Stankovic A 1988 Real-time computing systems: The next generation, COINS TR, University of Massachussetts

Shyamasundar R K 1991a Real-time programming languages: A survey, Lecture Notes, Workshop on Real-Time Embedded Computing Systems, Bangalore

Shyamasundar R K 1991b Specification and verification of real-time systems, Lecture Notes, Workshop on Real-Time Embedded Computing Systems, Bangalore

Shyamasundar R K, Narayana K T, Pitssi T 1987 Semantics of nondeterministic asynchronous broadcast networks, *ICALP'87*

Shyamasundar R K, Patterson A, Agha G 1991 An actor-based framework for concurrent object oriented programming, *Proceedings of Baastad Workshop on Concurrency, Sweden*

Wirth N 1977 Towards a discipline of real-time programming. *Commun. ACM* 20: 577–583

Zwiers J 1988 *Compositionality, concurrency and partial correctness: Proof theories of processes and their connection*, Ph D thesis, Eindhoven University of Technology, Eindhoven